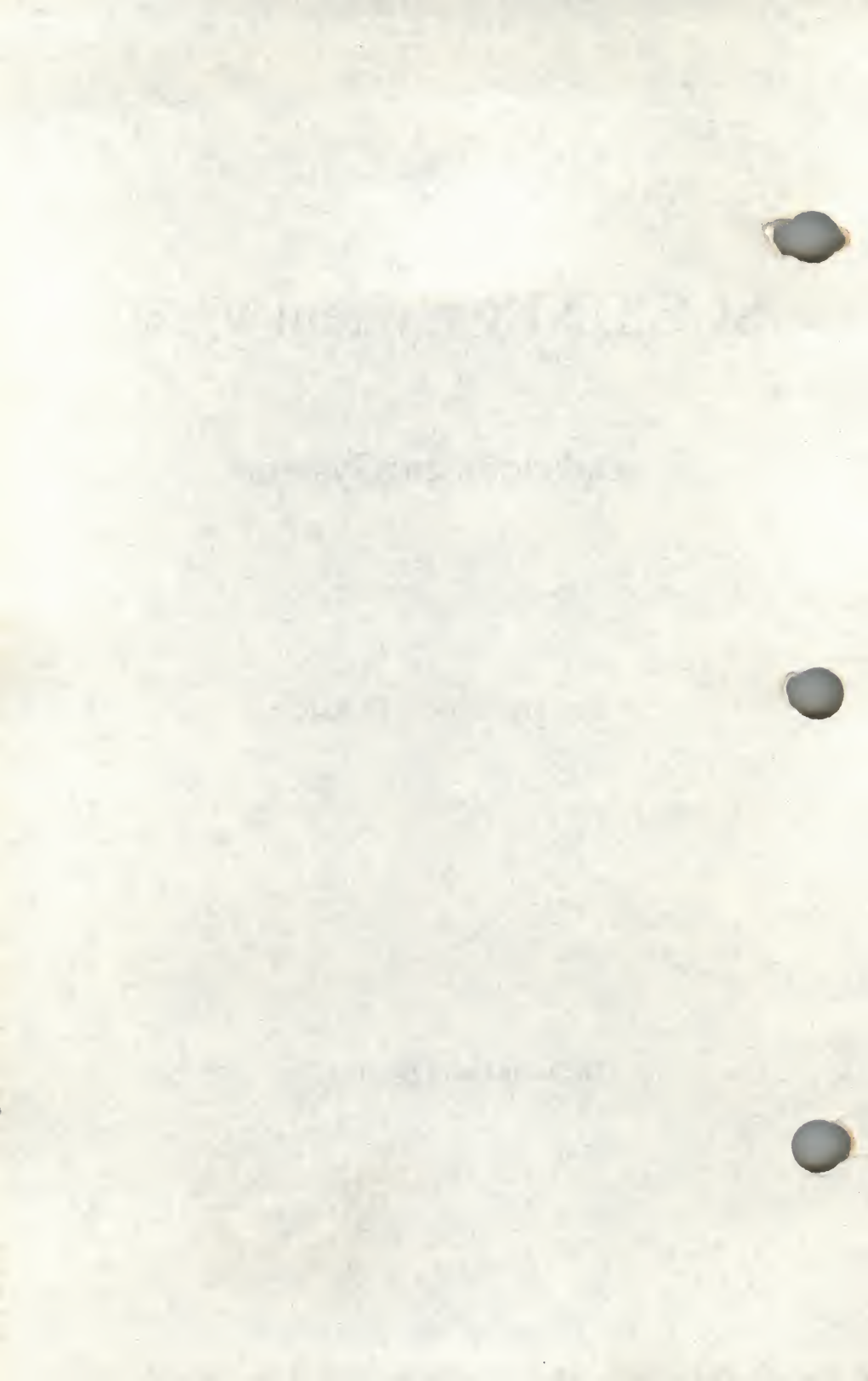


SCO UNIX[®] System V/386

Development System

Programmer's Guide

The Santa Cruz Operation, Inc.



Portions © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989 Microsoft Corporation.

All rights reserved.

Portions © 1989 AT&T.

All rights reserved.

Portions © 1983, 1984, 1985, 1986, 1987, 1988, 1989 The Santa Cruz Operation, Inc.

All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95062, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) OF THE COMMERCIAL COMPUTER SOFTWARE -- RESTRICTED RIGHTS CLAUSE AT FAR 52.227-19 OR SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. "CONTRACTOR/ MANUFACTURER" IS THE SANTA CRUZ OPERATION, INC., 400 ENCINAL STREET, P.O. BOX 1900, SANTA CRUZ, CALIFORNIA, 95061, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.

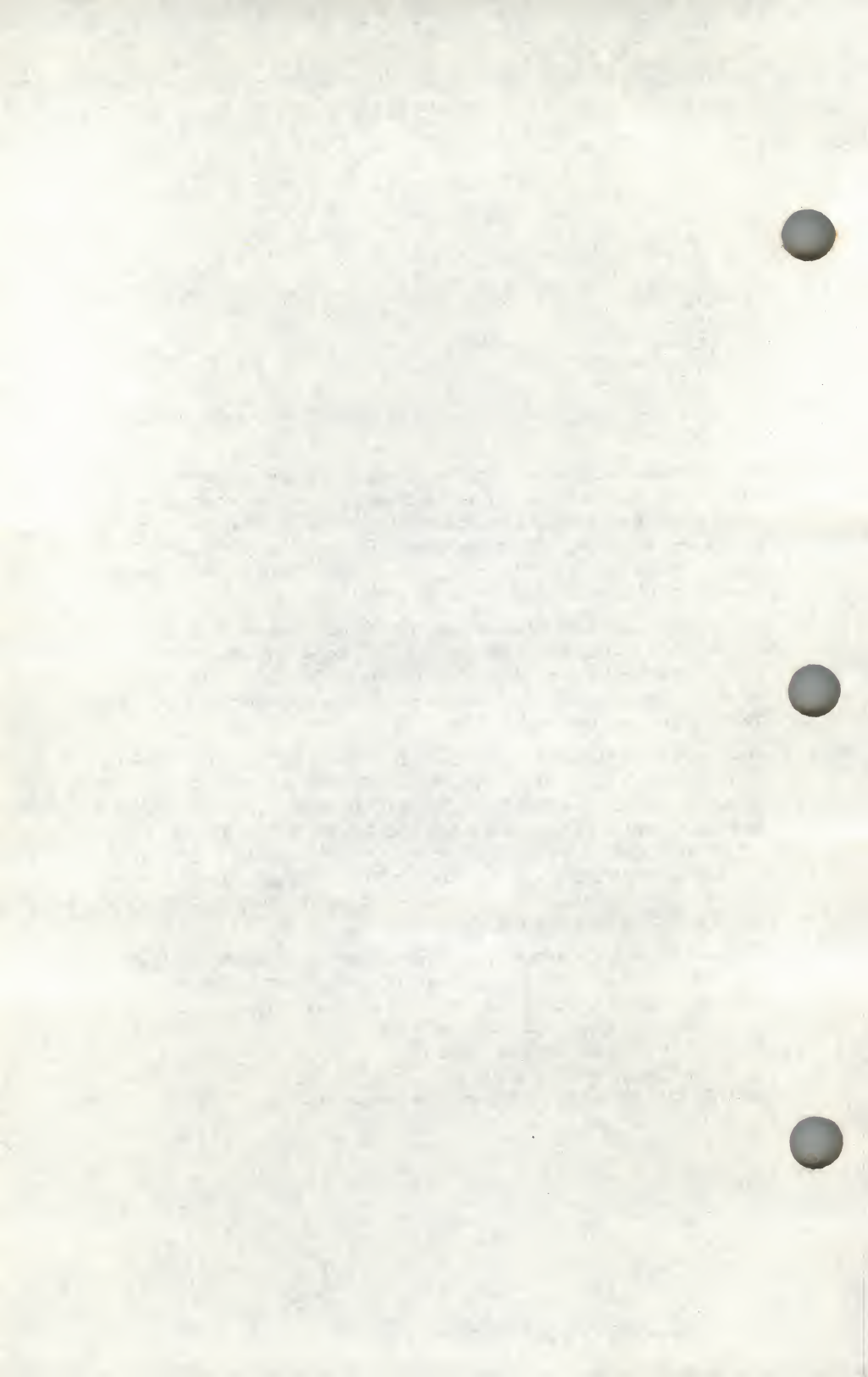
DOCUMENTER'S WORKBENCH is a registered trademark of AT&T.

Intel is a registered trademark of Intel Corporation.

TELETYPE is a registered trademark of AT&T.

UNIX is a registered trademark of AT&T.

WRITER'S WORKBENCH is a registered trademark of AT&T.



Contents

1 Introduction

- Introduction 1-1
- Creating Programs 1-2
- Creating and Maintaining Libraries 1-3
- Maintaining Program Source Files 1-4
- Creating Programs with Shell Commands 1-5
- Using This Guide 1-6
- Notational Conventions 1-8
- Referencing Commands 1-9

2 lex

- An Overview of **lex** Programming 2-1
- Writing **lex** Programs 2-3

3 lint

- Introduction 3-1
- Usage 3-2
- lint** Message Types 3-4

4 yacc

- Introduction 4-1
- Basic Specifications 4-4
- Parser Operation 4-12
- Ambiguity and Conflicts 4-17
- Precedence 4-23
- Error Handling 4-27
- The **yacc** Environment 4-30
- Hints for Preparing Specifications 4-32
- Advanced Topics 4-36

5 make

- Introduction 5-1
- Basic Features 5-2
- Description Files and Substitutions 5-6
- Recursive **Makefiles** 5-9
- The Tilde in **SCCS** File Names 5-16
- Command Usage 5-20

6 The Link Editor

The Link Editor 6-1
Link Editor Command Language 6-4
Notes and Special Considerations 6-20

7 Source Code Control System (SCCS)

Introduction 7-1
SCCS for Beginners 7-2
Delta Numbering 7-7
SCCS Command Conventions 7-10
SCCS Commands 7-12

8 m4: A Macro Processor

Introduction 8-1
Invoking m4 8-2
Defining Macros 8-3

9 C Programmer's Productivity Tools

Introducing the C Programmer's Productivity Tools 9-1
cscope 9-3
lprof 9-26
Profiling Examples 9-41

10 Extended Terminal Interface

Overview 10-1
What is ETI? 10-5
Basic ETI Programming 10-8
Simple Input and Output 10-17
Windows 10-53
Panels 10-64
Compiling and Linking Panel Programs 10-65
Creating Panels 10-66
Elementary Panel Window Operations 10-67
Moving Panels to the Top or Bottom of the Deck 10-70
Updating Panels on the Screen 10-71
Making Panels Invisible 10-73
Fetching Panels Above or Below Given Panels 10-76
Setting and Fetching the Panel User Pointer 10-78
Deleting Panels 10-81
Menus 10-82
Compiling and Linking Menu Programs 10-83
Overview: Writing Menu Programs in ETI 10-84
Creating and Freeing Menu Items 10-88

- Two Kinds of Menus: Single- and Multi-Valued 10-90
- Manipulating Item Attributes 10-92
- Setting the Item User Pointer 10-96
- Creating and Freeing Menus 10-99
- Manipulating Menu Attributes 10-101
- Displaying Menus 10-104
- Menu Driver Processing 10-120
- Manipulating the Menu User Pointer 10-141
- Setting and Fetching Menu Options 10-143
- Forms 10-146
- Compiling and Linking Form Programs 10-147
- Overview: Writing Form Programs in ETI 10-148
- Creating and Freeing Fields 10-154
- Manipulating Field Attributes 10-158
- Setting the Field Foreground, Background, and Pad Character 10-169
- Some Helpful Features of Fields 10-171
- Manipulating Field Options 10-177
- Creating and Freeing Forms 10-181
- Manipulating Form Attributes 10-184
- Displaying Forms 10-187
- Form Driver Processing 10-195

11 sdb: The Symbolic Debugger

- Introduction 11-1
- Using **sdb** 11-2

12 adb: A Program Debugger

- Introduction 12-1
- Starting and Stopping **adb** 12-2

13 File and Record Locking

- Introduction 13-1
- Terminology 13-2
- File Protection 13-4
- Selecting Advisory or Mandatory Locking 13-16

14 Shared Libraries

- Introduction 14-1
- Using a Shared Library 14-2
- Building a Shared Library 14-13
- Summary 14-52

15 Interprocess Communication

Introduction 15-1
Messages 15-2
Semaphores 15-32

16 Common Object File Format (COFF)

The Common Object File Format (COFF) 16-1

17 Programming on a Secure System

Programming On A Secure System 17-1

A Glossary

Glossary A-1

Chapter 1

Introduction

Introduction 1-1

Creating Programs 1-2

Creating and Maintaining Libraries 1-3

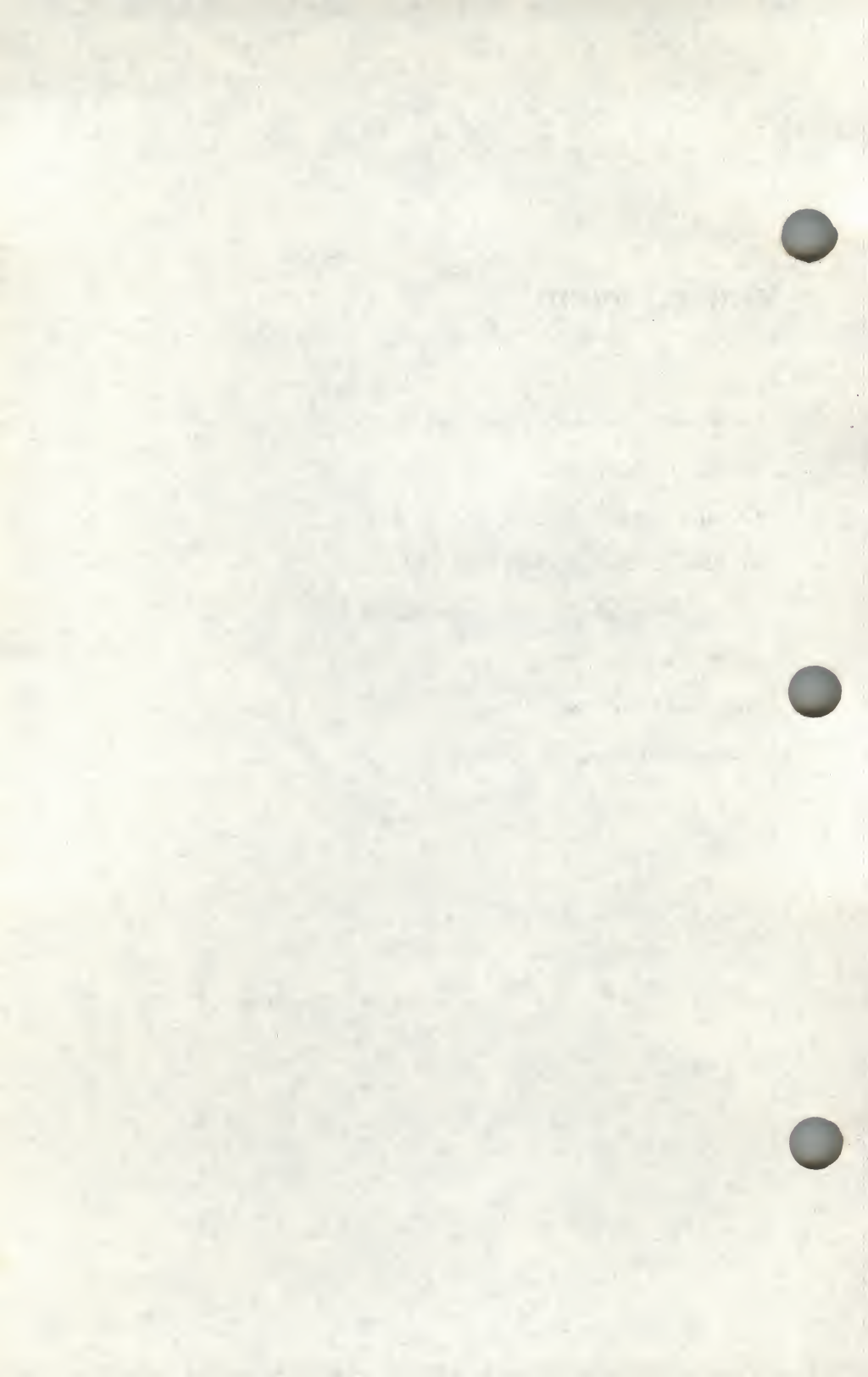
Maintaining Program Source Files 1-4

Creating Programs with Shell Commands 1-5

Using This Guide 1-6

Notational Conventions 1-8

Referencing Commands 1-9



Introduction

This guide explains how to use the UNIX Software Development System to create and maintain C language and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to:

- create C and assembly language programs for execution on the UNIX system
- debug programs
- automatically create C and assembly language
- maintain different versions of the programs that you develop

The following sections introduce the programs and commands of the UNIX Software Development System.

1 Creating Programs

The C programming language can meet the needs of most programming projects. A complete description of how to write, compile, link, and run C programs under the UNIX operating system is provided in the following guides:

- *UNIX C User's Guide*
- *UNIX C Language Reference*
- *UNIX C Library Guide*

You can also create assembly language programs using the UNIX macro assembler **masm**. It assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**, the UNIX linker. **ld** links relocatable object files created by the C compiler or assembler to produce executable programs. Note that the **cc** command invokes the linker and the assembler automatically, so use of either **masm** or **ld** is optional. For a complete description of how to write, compile, link, and run assembly programs under the UNIX operating system, see the *UNIX Macro Assembler*.

You can create source files for lexical analyzers and parsers using the program generators **lex** and **yacc**. You use lexical analyzers in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. You use parsers in programs to convert meaningful sequences of tokens and values into actions. The UNIX **lex** program generates lexical analyzers, written in C program statements, from given specification files. The UNIX **yacc** program generates parsers, written in C program statements, from given specification files. You can use **lex** and **yacc** together to make complete programs.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read “lex: A Lexical Analyzer” and “yacc: A Compiler-Compiler,” for explanations of the **lex** and **yacc** program generators.

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files, using the **m4** macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file. For more information, see “m4: A Macro Processor.”

Creating and Maintaining Libraries

1

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib** programs. **ar**, the UNIX archiver, creates libraries of relocatable object files. The UNIX random library generator **ranlib**, converts archive libraries to random libraries and places a table of contents at the front of each library. For more information on **ar**, see the *C User's Guide*. For more information on **ranlib**, see the *C Library Guide*.

Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS (Source Code Control) commands. The **make** program is described in “make: A Program Maintainer,” and the SCCS commands are described in “SCCS: A Source Code Control System.”

The **make** program is the UNIX program maintainer. It automates the steps required to create executable programs and provides a mechanism for ensuring that programs are up-to-date. You use **make** with medium-scale programming projects.

The Source Code Control (SCCS) commands let you maintain different versions of a single program. The commands compress all versions of a source file into a single file containing a list of differences. These commands also restore compressed files to their original size and content.

Many UNIX commands let you carefully examine a program's source files. The **ctags** command creates a *tags* file so that C functions can be quickly found in a set of related C source files. The **mkstr** command creates an error message file by examining a C source file.

Creating Programs with Shell Commands

1

In some cases, it is easier to write a program as a series of UNIX shell commands than it is to create a C language program. Shell commands provide much of the same control capability as the C language, and give direct access to all the commands and programs normally available to the UNIX user.

The **csh** command invokes the C-shell, a UNIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has a facility for creating aliases, and a command history feature. For more information, see "The C-Shell."

Using This Guide

This guide is intended for programmers who are familiar with the C programming language, the assembly programming language, and with the UNIX system. The following list briefly describes each chapter.

Chapter 1, “Introduction,” introduces the UNIX Software Development programs provided with this package.

Chapter 2, “lex: A Lexical Analyzer,” explains how to create lexical analyzers using the program generator **lex**.

Chapter 3, “lint: A C Program Checker,” describes the UNIX program checker, **lint**, and describes the available options.

Chapter 4, “yacc: A Compiler-Compiler,” explains how to create parsers using the program generator **yacc**.

Chapter 5, “make: A Program Maintainer,” explains how to automate the development of a program or project using the **make** program.

Chapter 6, “ld: the Link Editor” describes the design and function of the UNIX link editor, **ld**. The available options are explained in detail.

Chapter 7, “SCCS: A Source Code Control System,” explains how to control and maintain all versions of a project’s source files using the SCCS commands. “adb: A Program Debugger,” explains how to debug C and assembly language programs using the UNIX debugger **adb**.

Chapter 8, “m4: A Macro Processor,” explains how to use, create, and process macros using the **m4** macro processor.

Chapter 9, “C Programmer’s Productivity Tools” discusses the various programs available to aid the C programmer in development.

Chapter 10, “Extended Terminal Interface” describes the ETI screen processing package for the *terminfo* terminal capability database.

Chapter 11, “sdb: The Symbolic Debugger,” explains how to debug C, assembly language and Fortran programs using the UNIX debugger **sdb**.

Chapter 12, “adb: A Debugger,” explains how to debug C and assembly language programs using **adb**.

Chapter 13, “File and Record Locking” describes the UNIX Development System’s file and record locking capabilities.

Chapter 14, “Shared Libraries” describes the shared C Libraries.

Chapter 15, “Interprocess Communication” describes the **ipc** interface for process communication.

Chapter 16, “Common Object File Format” describes the COFF object file format.

Chapter 17, “Programming in a Secure Environment” deals with issues pertaining to programming for systems where security is an important issue.

Appendix A, “Glossary” gives a list of key terms and their meanings and history.

C language programmers should read the *UNIX C User’s Guide* for an explanation of how to compile and debug C language programs.

Assembly language programmers should read the *UNIX Macro Assembler User’s Guide* for an explanation of how to compile and debug **masm** programs.

1 Notational Conventions

This guide uses a number of special symbols to describe the syntax of UNIX commands. The following is a list of these symbols and their meaning.

Examples	Examples of program fragments or commands are indented and set in monospace type.
<code>monospace</code>	Monospace type is used for sample command-lines, program code and examples, and sample sessions.
SMALL	Small capitals indicate keynames, constants, or error conditions.
bold	Boldface characters indicate a command or program name, any command option or flag, and any function, routine, or subroutine.
<i>italics</i>	Italic characters indicate a filename (for example, <i>/etc/ttys</i>) or a placeholder for a command argument. When typing a command, replace a placeholder with an appropriate filename, number, or option. Italics are also used to give emphasis in the text, and are used to identify the first use of a technical term.

Referencing Commands

Within the *UNIX Programmer's Guide*, a command may end with one of the following letters in parentheses:

(S), (F), (M), (CP), (C), or (ADM)

These notations mark which section of the *UNIX Reference* you will find a command in:

(S)	System calls
(F)	Files and Formats
(M)	Miscellaneous
(CP)	Programming Commands
(C)	Commands
(ADM)	System Administration

Chapter 2

lex

· An Overview of **lex** Programming 2-1

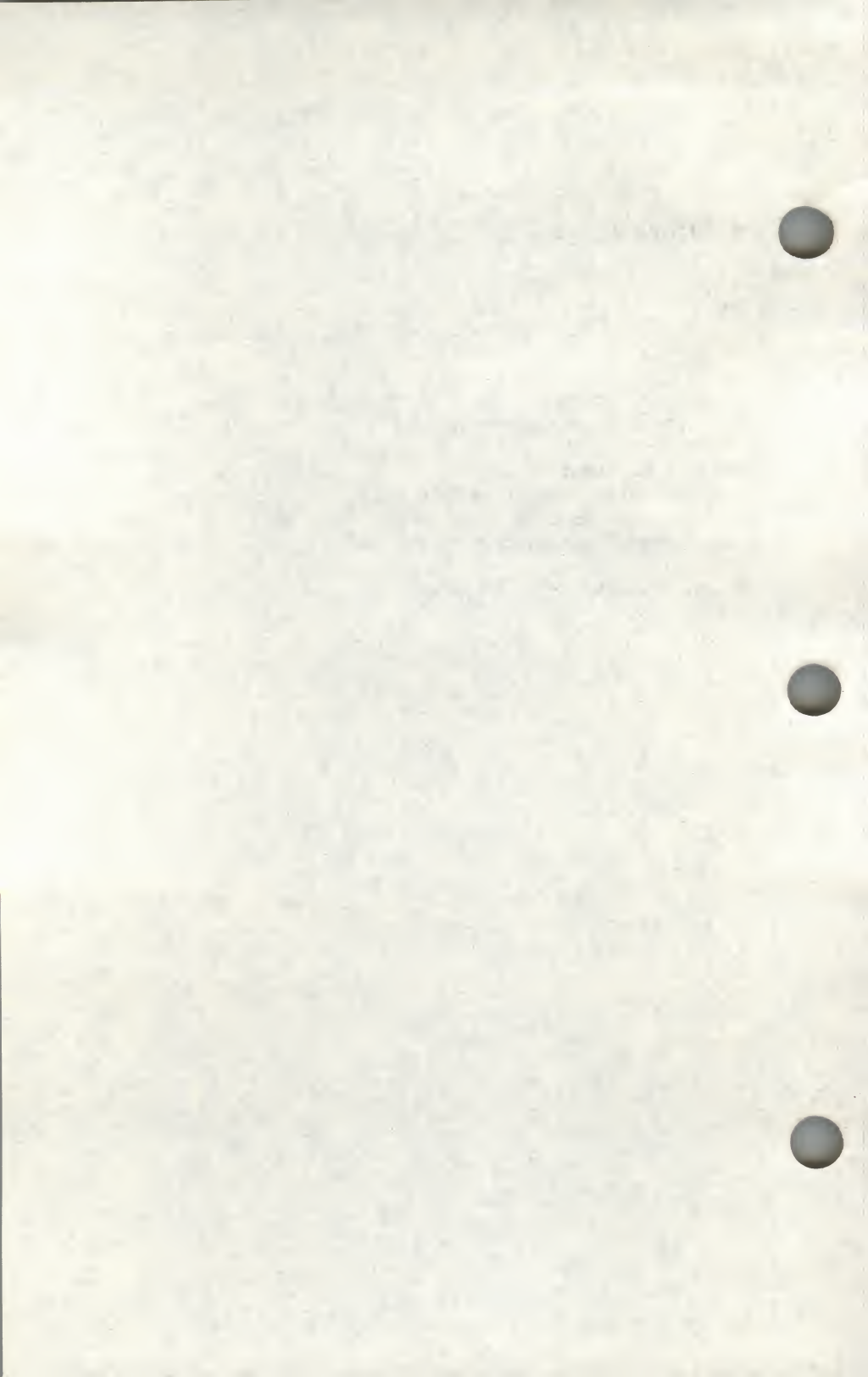
Writing **lex** Programs 2-3

 The Fundamentals of **lex** Rules 2-3

 Advanced **lex** Usage 2-7

 Using **lex** with **yacc** 2-15

Running **lex** under the UNIX System 2-17



An Overview of `lex` Programming

The software tool `lex` lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled. The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. In the compiler writing case, creating the ability to solve the problem requires implementing the compiler's lexical analyzer; hence the name `lex`.

It is not essential to use `lex` to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what `lex` does is produce such C programs. (`lex` is therefore called a program generator.) What `lex` offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using `lex` considerably outweigh it.

To understand what `lex` does, see the diagram in Figure 2-1. We begin with the `lex` source (often called the `lex` specification) that you, the programmer, write to solve the problem at hand. This `lex` source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the `lex` program generator. The output of the program generator is a C program that, in turn, must be compiled by a host language C compiler in order to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic—user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

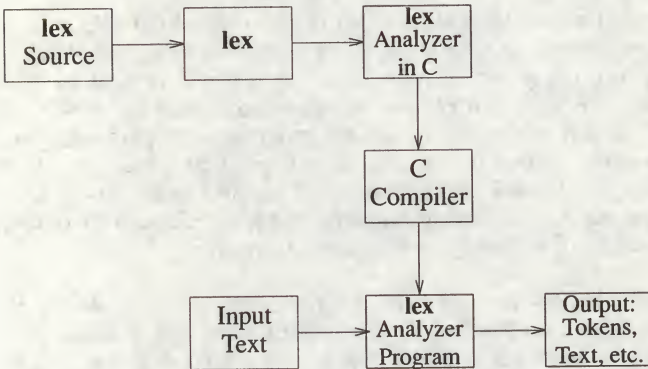
An Overview of lex Programming

lex can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see

- how to write **lex** source to do some of these tasks
- how to translate **lex** source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that **lex** provides.

Figure 2-1 Creation and Use of a Lexical Analyzer with **lex**



Writing lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter `% %`. If a subroutines section follows, another `% %` delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term specification—it may mean either the entire **lex** source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, **lex** writes out the input exactly as it finds it. So, the simplest **lex** program is just the beginning rules delimiter, `% %`. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in with a notation called regular expressions. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all. For example,

```
apple
orange
pluto
```

These three regular expressions match any occurrences of those character strings in an input text. If you want to have your lexical analyzer, **a.out**, remove every occurrence of **orange** from the input text, you could specify the rule

```
orange;
```

Writing lex Programs

Because you did not specify an action on the right (before the semicolon), **lex** does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string **orange** at all.

2 Unlike **orange** above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite. Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The **+** operator, for instance, means one or more occurrences of the preceding expression, the **?** means 0 or 1 occurrence of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and ***** means 0 or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. We will see an example in a moment.) Therefore, **m+** is a regular expression matching any string of **ms** such as each of the following:

```
mnm  
m  
mmmmmm  
mm
```

In like manner, **7*** is a regular expression matching any string of zero or more **7s**:

```
77  
77777  
  
777
```

The string of blanks on the third line matches simply because it has no **7s** in it at all.

Brackets, **[]**, indicate any one character from the string of characters specified between the brackets. Thus, **[dgka]** matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma within the brackets would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen, **-**. The sequence **[a-z]**, for instance, indicates any lowercase letter. Somewhat more interestingly,

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether uppercase or lowercase), any digit, an asterisk, an ampersand, or a sharp character.

Given the input text

```
$$$$?? ?????!!*$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits. That is just what the regular expression says. The first pair of brackets matches any letter. The second pair, if it were not followed by a *, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the asterisk, *, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

```
e
pay
distance
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers:

```
not_idenTIFER
5times
$hello
```

because **not_idenTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **\$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them, as characters to look for, in our search pattern. The last example, for instance, will not recognize text with an * in it. **lex** solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a \ is taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, an * followed by any

Writing lex Programs

number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a `\` itself, we need two backslashes: `\\`

2

Actions

Once `lex` recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. Kinds of actions include recording the token type found and its value, if any, replacing one token with another, and counting the number of instances of a token or token type. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression *Amelia Earhart* and to note such recognition, the rule

```
"Amelia Earhart"    printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. `lex` uses the standard escape sequences from C like `\n` for end-of-line. To count lines we might have

```
\n    lineno++;
```

where `lineno`, like other C variables, is declared in the definitions section that we will discuss later.

`lex` stores every character string that it recognizes in a character array called `yytext[]`. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you may choose to) write it on several lines. To inform `lex` that the action is for one rule only, simply enclose the C code in braces.

For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum), and print out each one as soon as it is found, your **lex** code might be

```
+?[1-9]+      { digstrngcount++;
                printf("%d\n",digstrngcount);
                printf("%s\n", yytext);  }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the **?** indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings, because that portion following the minus sign, **-**, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

The **lex** command provides a suite of features that lets you process input text riddled with quite complicated patterns. These include rules that decide what specification is relevant, when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining an example drawing together several of the points already covered.

```
%%
-[0-9]+      printf("negative integer");
+?[0-9]+      printf("positive integer");
-0.[0-9]+     printf("negative fraction, no whole number part");
rail[ ]+road  printf("railroad is one word");
crook         printf("Here's a crook");
function      subprogcount++;
G[a-zA-Z]*    { printf("may have a G word here: ", yytext);
                Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and -1. Use of the terminating **+** in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for **railroad** matches cases where one or more blanks intervene between the two syllables of the word. In the cases of **railroad** and **crook**, you may have simply printed a synonym rather than the messages stated. The rule recognizing a **function** increments a counter.

Writing lex Programs

The last rule illustrates several points:

- The braces specify an action sequence extending over several lines.
- Its action uses the **lex** array **yytext[]**, which stores the recognized character string.
- Its specification uses the ***** to indicate that zero or more letters may follow the **G**.

2

Some Special Features

Besides storing the recognized character string in **yytext[]**, **lex** automatically counts the number of characters in a match and stores it in the variable **yylen**. You may use this variable to refer to any specific character just placed in the array **yytext[]**. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) in a just recognized integer, you might write

```
[1-9]+      {if (yylen > 2)
              printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. *Prima facie*, any reserved word, for instance, could match two rules. In the lexical analyzer example developed later in the section on **lex** and **yacc**, the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

Note

lex follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize `>` and `>=`. If the text has the string `>=` at one point, you might worry that the lexical analyzer would stop as soon as it recognized the `>` character to execute the rule for `>` rather than read the next character and execute the rule for `>=`.

2

Note

`lex` matches the longest character string possible and executes the action for that.

Here it would recognize the `>=` and act accordingly. As a further example, the rule would enable you to distinguish `+` from `++` in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you have in fact found the string until you have read the additional characters. These cases reveal the importance of trailing context. The classic example here is the `DO` statement in FORTRAN. In the statement

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index `k` until we read the first comma. Until then, we might have the assignment statement

```
DO50k = 1
```

(FORTRAN ignores all blanks.) The way to handle this is to use the forward-looking slash, `/` (not the backslash, `\`), which signifies that what follows is trailing context, something not to be stored in `yytext[]`, because it is not part of the token itself. So the rule to recognize the FORTRAN `DO` statement could be

```
30/[ ]*[0-9][ ]*[a-zA-Z0-9]+=[a-zA-Z0-9]+, printf("foundDO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

Writing lex Programs

lex uses the **\$** as an operator to mark a special trailing context—the end of line. (It is therefore equivalent to **\n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

```
[ \t]+$ ;
```

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, **^**, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ] printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks—**input()**, **unput(c)**, and **output(c)**, respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input()**, thus:

```
\ " while (input() != '"');
```

Upon finding the first double quotation mark, the generated lexical analyzer will simply continue reading all subsequent characters, so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you may use standard I/O routines in C to rewrite the functions **input()**, **unput(c)**, and **output**. These and other programmer-defined functions should be placed in your subroutine section. Your new routines will then replace the standard ones. The standard **input()**, in fact, is equivalent to **getchar()**, and the standard **output(c)** is equivalent to **putchar(c)**.

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include **yymore()**, **yyless(n)**, and **REJECT**. Recall that the text matching a given specification is stored in the array **yytext[]**. In general, once the action is performed for the specification, the characters in **yytext[]** are overwritten with succeeding characters in the input stream to form the next match. The function **yymore()**, by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext[]**. This lets you do one thing and then another, when one string of characters is significant and a longer one, which includes the first, is significant as well.

Consider a character string bound by **B**s and interspersed with one at an arbitrary location.

B...B...B

In a simple code-deciphering situation, you may want to count the number of characters between the first and second **B**'s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is

```
B[~B]*      { if (flag = 0)
                save = yyleng;
                flag = 1;
                yymore();
            else
            {
                importantno = save + yyleng;
                flag = 0; }
            }
```

where **flag**, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyles(n)** lets you reset the end point of the string to be considered to the *n*th character in the original **yytext[]**. Suppose you are again in the code-deciphering business, and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say upper- or lowercase **Z**. The code you want might be

```
[a-zA-Y]+[Zz]    { yyles(yyleng/2);
                    ... process first half of string... }
```

Finally, the function **REJECT** lets you more easily process strings of characters. **REJECT** does this by immediately jumping to the next rule and its specification without changing the contents of **yytext[]**. If you want to count the number of occurrences both of the regular expression **snapdragon** and of its subexpression **dragon** in an input text, the following will do:

```
snapdragon      {countflowers++; REJECT;}
dragon           countmonsters++;
```


Writing lex Programs

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions **comedian** and **diana**, even where the input text has sequences such as **comediana.:**

```
comedian      {comiccount++; REJECT;}  
diana        princesscount++;
```

2

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, the counters and other necessary variables are declared in the definitions section commencing the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later:

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it will be accessible in the routine, say a parser, that calls it. If, on the other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself, right after the left brace, { .

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file, to be included in every file that needs them.

One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#defines** for token names. Like the declarations, **#include** statements should come between **%{** and **%}**, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the **%}** that ends your **#include**'s and declarations, place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you use abbreviations in your rules, enclose them within braces.

Note

The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:

```
D          [0-9]
L          [a-zA-Z]
B          [ ]
%%
-{D}+      printf("negative integer");
+?{D}+     printf("positive integer");
-0.{D}+    printf("negative fraction");
G{L}*      printf("may have a G word here");
rail{B}+road printf("railroad is one word");
crook      printf("criminal");
"\."/ {B}+ printf(".\"");
.          .
```

The last rule, newly added to the example and somewhat more complex than the others, ensures that a period always precedes a quotation mark at the end of a sentence. It would change **example"** to **example."**

2

2

2

2

2

2

2

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX System program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read this section because it covers information of interest to all **lex** programmers.

The lexical analyzer is named **yylex()**. This name is convenient because **yacc** calls its lexical analyzer by this very name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want to end each **lex** action with the statement **return token**, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called **y.tab.c** by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >. Consider the following portion of **lex** source for a lexical analyzer for some programming language perhaps slightly reminiscent of Ada:

```
begin          return(BEGIN);
end            return(END);
while          return(WHILE);
if             return(IF);
package        return(PACKAGE);
reverse        return(REVERSE);
loop           return(LOOP);
[a-zA-Z][a-zA-Z0-9]* { tokval = put_in_tabl();
                      return(IDENTIFIER); }
[0-9]+         { tokval = put_in_tabl();
                      return(INTEGER); }
\+            { tokval = PLUS;
                      return(ARITHOP); }
-             { tokval = MINUS;
                      return(ARITHOP); }
>             { tokval = GREATER;
                      return(RELOP); }
>=            { tokval = GREATEREQUAL;
                      return(RELOP); }
```


Writing lex Programs

Despite appearances, the tokens returned and the values assigned to **tokval**, are indeed integers. Good programming style dictates that we use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C. For example,

```
#define BEGIN 1
#define END 2
.
#define PLUS 7
.
```

If the need arises to change the integer for some token type, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to insert the statement

```
#include y.tab.h
```

into the definitions section of your **lex** source. The file **y.tab.h** provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, a function **put_in_tabl()** places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl()** assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl()** would be a routine that the compiler writer might place in the subroutines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable **PLUS**, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a + sign is recognized, the action assigns to **tokval** the value 7, which indicates the +. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by **ARITHOP** or **RELOP**).

Running lex under the UNIX System

As you review the following few steps, you might recall Figure 2-1 at the start of the chapter. To produce the lexical analyzer in C, run

```
lex lex.l
```

where **lex.l** is the file containing your **lex** specification. The name **lex.l** is conventionally the favorite, but you can use any name you like. The output file that **lex** produces is automatically called **lex.yy.c**; this is the lexical analyzer program that you created with **lex**. You then compile and link this as you would any C program, making sure that you invoke the **lex** library with the **-ll** option:

```
cc lex.yy.c -ll
```

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you need not supply your own **main()**.

If you have the **lex** specification spread across several files, you can run **lex** with each of them individually, but be sure to rename or move each **lex.yy.c** file (with **mv**) before you run **lex** on the next one. Otherwise, each will overwrite the previous one. Once you have all the generated **.c** files, you can compile all of them, of course, in one command line.

With the executable lexical analyzer produced, you are ready to analyze any desired input text. Suppose that the text is stored under the file name **textin** (this name is arbitrary). The lexical analyzer by default takes input from your terminal. To have it take the file **textin** as input, use redirection, thus:

```
a.out < textin
```

where **a.out** is the executable lexical analyzer.

By default, output will appear on your terminal. You can redirect this as well:

```
a.out < textin > textout
```

Running lex under the UNIX System

In running **lex** with **yacc**, either may be run first. For example,

```
yacc -d grammar.y
lex lex.l
```

creates a parser in the file **y.tab.c**. (The **-d** option creates the file **y.tab.h**, which contains the **#define** statements that associate the **yacc**-assigned integer token values with the user-defined token names.) To compile and link the output files produced, run

```
cc lex.yy.c y.tab.c -ly -ll
```

Note that the **yacc** library is loaded (with the **-ly** option) before the **lex** library (with the **-ll** option) to ensure that the **main()** program supplied will call the **yacc** parser.

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the file name argument. If you care to see the C program, **lex.yy.c**, that **lex** generates on your terminal (the default output device), use the **-t** option.

```
lex -t lex.l
```

The **-v** option prints out for you a small set of statistics describing the so-called finite automata that **lex** produces with the C program **lex.yy.c**. (For a detailed account of finite automata and their importance for **lex**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 500. If your **lex** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing the following entry in the definitions section of your **lex** source;

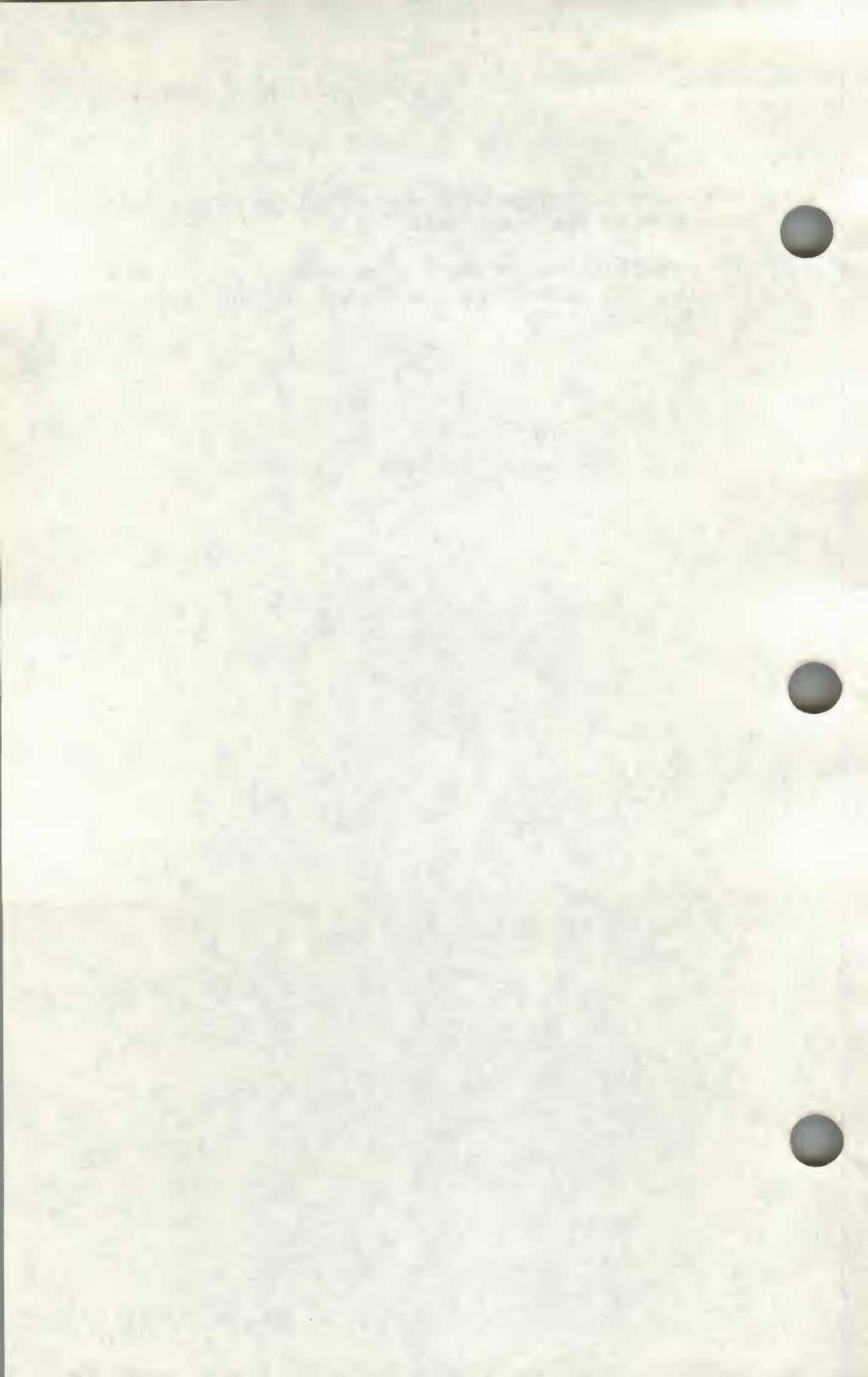
```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. (The **-v** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *Programmer's Reference* page on `lex` for a list of all the options available with the `lex` command.

This tutorial has introduced you to `lex` programming. As with any programming language, the way to master it is to write programs and then write some more.



Chapter 3

lint

Introduction 3-1

Usage 3-2

lint Message Types 3-4

Unused Variables and Functions 3-4

Set/Used Information 3-5

Flow of Control 3-6

Function Values 3-6

Type Checking 3-8

Type Casts 3-9

Nonportable Character Use 3-9

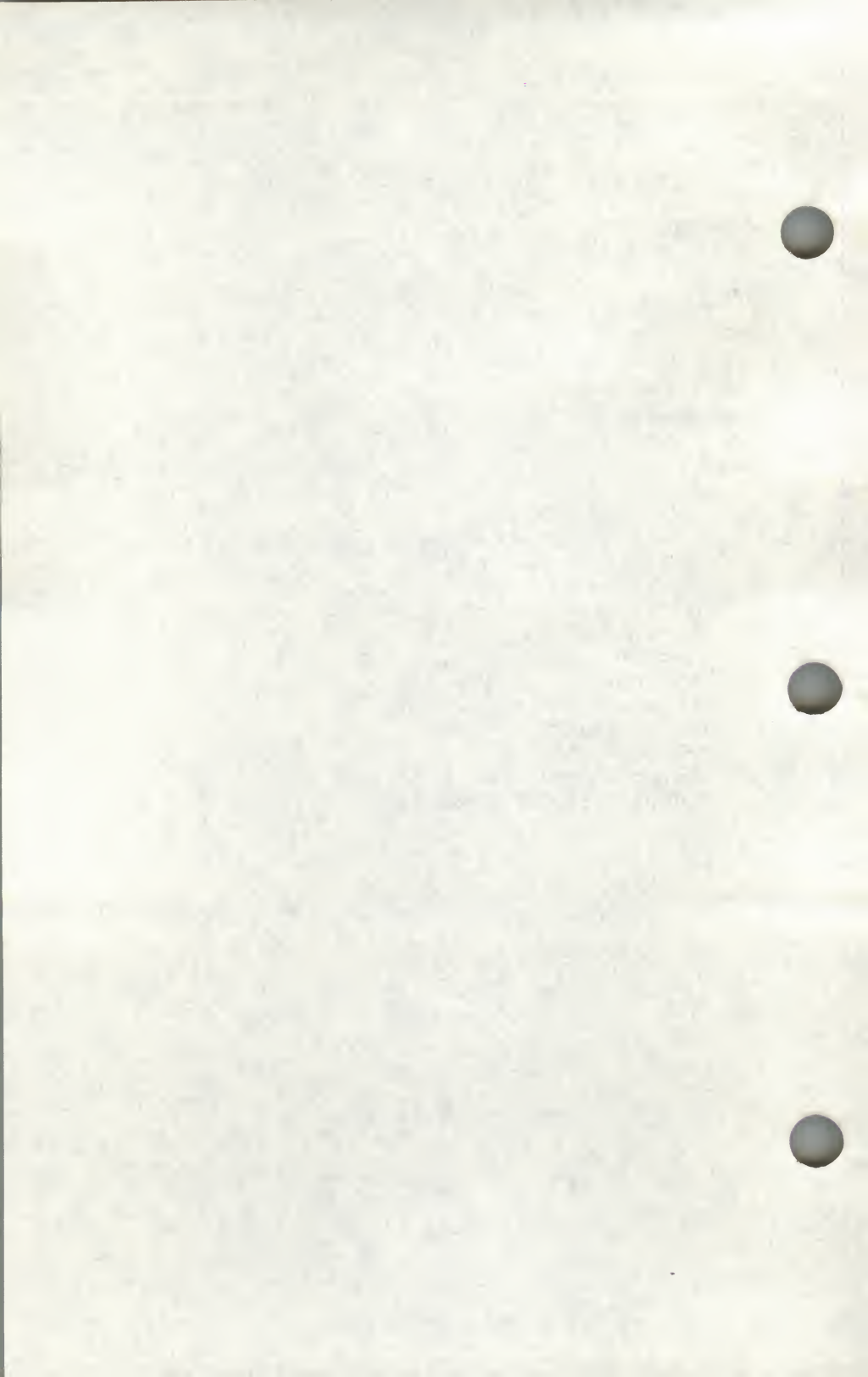
Assignments of **longs** to **ints** 3-10

Strange Constructions 3-10

Old Syntax 3-11

Pointer Alignment 3-12

Multiple Uses and Side Effects 3-12



Introduction

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than does the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects legal C constructs which are wasteful or error prone. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [*options*] *files ... library-descriptors ...*

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

3

The options that are currently supported by the **lint** command are the following:

- a** suppresses messages about assignments of long values to variables that are not long
- b** suppresses messages about break statements that cannot be reached
- c** checks only for intra-file bugs; leaves external information in files suffixed with **.ln**
- h** does not apply heuristics (which attempt to detect bugs, improve style, and reduce waste)
- n** does not check for compatibility with either the standard or the portable **lint** library
- o name** creates a lint library from input files named **llib-lname.ln**
- p** checks portability
- u** suppresses messages about function and external variables used and not defined or defined and not used
- v** suppresses messages about unused arguments in functions
- x** does not report variables referred to by external declarations but never used

When more than one option is used, they should be combined into a single argument, such as **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix `.c`, which is mandatory for **lint** and the C compiler.

The **lint** command accepts certain arguments, such as:

-lm

These arguments specify libraries that contain C functions. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These descriptor files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are: the declaration of the function return type; whether the dummy function returns a value; and the number and types of arguments to the function. The `VARARGS` and `ARGSUSED` comments can be used to specify features of the library functions. The next section, “**lint** Message Types,” describes how this is done.

The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file but not used in a source file do not result in messages. The **lint** command does not simulate a full library-search algorithm, and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **-p** option is used, another file containing descriptions of the standard library routines is checked which are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

lint Message Types

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

3 As sets of programs evolve and develop, previously-used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** command prints messages about variables and functions which are defined but not otherwise mentioned, unless a message is suppressed by means of the **-u** or **-x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks, depending on which arguments are used. Normally, **lint** prints messages about unused arguments; however, the **-v** option is available to suppress the printing of these messages. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
· /* ARGSUSED */
```

to the source code before the function. This has the effect of the **-v** option for only one function.

Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about a variable number of arguments in calls to a function. The comment should be added before the function definition.

In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit representing the number of arguments that should be checked. For example,

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some, but not all, files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **-u** option suppresses the spurious messages.

Set/Used Information

The **lint** command attempts to detect cases where a variable is used before it is set. It detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. The **lint** command can print error messages about program fragments that are legal, but would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

The **lint** command attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom, and to recognize the special cases **while(C)** and **for(;;)** as infinite loops. Also, **lint** prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreachd portions of the program, use the **-b** option.

3

The **lint** command has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code that **lint** does not detect. The most serious effects of this are in the determination of returned function values. (See "Function Values.") If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment:

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **-b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. The **lint** command addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both:

```
return( expr );
```

and:

```
return ;
```

statements is cause for alarm; **lint** will give the message:

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
    if ( a ) return ( 3 );
    g ();
}
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced, when in fact nothing is wrong. A comment:

```
/*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
(void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (such as not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is serious.

Type Checking

The **lint** command enforces the type-checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure-selection operators
- between the definition and uses of functions
- in the use of enumerations

There are a number of operators that have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of **xs** can, of course, be intermixed with pointers to **xs**.

The type-checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function-argument and return-value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must be the same types as their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are **=**, initialization, **==**, **!=**, function arguments and return values.

If you want to turn off strict type checking for an expression, the comment:

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where **p** is a character pointer. The **lint** command will print a message as a result of detecting this. Consider the assignment:

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled those intentions. Nevertheless, **lint** will continue to print messages about this.

3

Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments being illegal or nonportable. For example, the fragment:

```
char c;
...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* to be an integer, because **getchar** is actually returning integer values. In any case, **lint** will print the message:

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the fields may be too small to hold the values. This is especially true because bit fields on some machines are considered signed quantities. While it may seem logical that a two-bit field declared type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of a **long** to an **int**, which will truncate the contents. This may happen in programs that have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. The **-a** option can be used to suppress messages about the assignment of **longs** to **ints**.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. It is hoped that the messages encourage better code quality and clearer style, and even point out bugs. The **-h** option is used to suppress these checks. For example, in the statement:

```
*p++ ;
```

the ***** does nothing. This provokes the message:

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;  
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test:

```
if( x > 0 ) ...
```

is equivalent to:

```
if( x != 0 )
```

which may not be the intended action. The **lint** command will print the message:

```
degenerate unsigned comparison
```

in these cases.

If a program contains something similar to:

```
if( 1 != 0 ) ...
```

lint will print the message:

```
constant in conditional context
```

because the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs that arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) ...
```

and:

```
x<<2 + 40
```

probably do not act as intended. The best solution is to parenthesize such expressions, and **lint** encourages this with an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (such as `+=`, `-=`, ...) could cause ambiguous expressions, like:

```
a -= 1 ;
```

which could be taken as either:

```
a -= 1 ;
```

or:

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity results from a macro substitution. The newer and preferred operators (such as `+=`, `-=`, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

lint Message Types

A similar issue arises with initialization. The older language allowed:

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties.

For example, the initialization:

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { ...
```

and the compiler must read past *x* to determine the correct meaning.

Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others, owing entirely to alignment restrictions. The **lint** command tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message:

```
possible pointer alignment problem
```

results from this situation.

Multiple Uses and Side Effects

With complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive.

Function calls embedded as arguments of other functions may or may not be treated comparably to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

So that the efficiency of C language on a particular machine is not unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the orders in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The **lint** command checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++];
```

will cause **lint** to print the message:

```
warning: i evaluation order undefined
```

to call attention to this condition.

Chapter 4

yacc

Introduction 4-1

Basic Specifications 4-4

 Actions 4-6

 Lexical Analysis 4-9

Parser Operation 4-12

Ambiguity and Conflicts 4-17

Precedence 4-23

Error Handling 4-27

The yacc Environment 4-30

Hints for Preparing Specifications 4-32

 Input Style 4-32

 Left Recursion 4-33

 Lexical Tie-Ins 4-34

 Reserved Words 4-35

Advanced Topics 4-36

 Simulating **error** and **accept** in Actions 4-36

 Accessing Values in Enclosing Rules 4-36

 Support for Arbitrary Value Types 4-37

 yacc Input Syntax 4-39

Examples 4-42

 1. A Simple Example 4-42

 2. An Advanced Example 4-45

Introduction

The **yacc** program provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes the following:

- a set of rules to describe the elements of the input
- a code to be invoked when a rule is recognized
- either a definition or a declaration of a low-level routine to examine the input

Then **yacc** turns the specification into a C language function that examines the input stream. This function, called a parser, works by calling the low-level input scanner. This is called a lexical analyzer. It picks up items from the input stream. The items selected are known as tokens. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule (an action) is invoked. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be:

```
date : month_name day ',' year ;
```

where **date**, **month_name**, **day**, and **year** represent constructs of interest; presumably, **month_name**, **day**, and **year** are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in evaluating the input. With proper definitions, the input:

```
July 4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates to the parser these as tokens. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

Introduction

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the rules:

```
month_name : 'J' 'a' 'n' ;  
month_name : 'F' 'e' 'b' ;  
...  
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month name and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule:

```
date : month '/' day '/' year ;
```

allowing:

```
7/4/1776
```

as a synonym for:

```
July 4, 1776
```

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan, input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter often can be corrected by making the lexical analyzer more powerful or by

rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructs that are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The remainder of this chapter describes the following subjects:

- the basic process of preparing a **yacc** specification
- the parser operation
- how to handle ambiguities
- how to handle operator precedences in arithmetic expressions
- error detection and recovery
- the operating environment and special features of the parsers **yacc** produces
- suggestions to improve the style and efficiency of the specifications
- advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to retain it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. Sections are separated by double percent signs (% %). (The single percent sign is generally used in **yacc** specifications as an escape character.)

A full specification file looks like this:

declarations
%%
rules
%%
subroutines

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest legal **yacc** specification is:

%%
rules

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules, each of which has the form:

A : BODY ;

where A represents a nonterminal symbol, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length and may be made up of letters, dots, underscores, and digits, although a digit may not be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ('). As in the C language, the backslash (\) is an escape character within literals, and all the C language escapes are recognized. Thus:

```
'\n'    newline
'\r'    return
'\''    single quote ( ' )
'\\'    backslash ( \ )
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  xxx in octal notation
```

are understood by **yacc**. The NULL character (\0 or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar (|) can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus the grammar rules:

```
A      : B C D ;
A      : E F  ;
A      : G   ;
```

can be given to **yacc** as:

```
A      : B C D
        | E F
        | G
        ;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by:

```
epsilon : ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon**.

Basic Specifications

Names representing tokens must be declared. This is most simply done by writing:

```
%token    name1  name2  ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

The start symbol has particular importance. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare explicitly the start symbol in the declarations section, using the **%start** keyword:

```
%start    symbol
```

4

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to, but not including, the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end-of-file or end-of-record.

Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. Actions may return values and obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and, as such, can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in curly braces, { and }.

The following examples are grammar rules with actions.

```
A      :  ' ( '  B  ' ) '
        {
            hello( 1, "abc" );
        }
```

```
XXX    :  YYY  ZZZ
        {
            (void) printf("a message\n");
            flag = 25;
        }
```

The dollar sign symbol (\$) is used to facilitate communication between the actions and the parser. The pseudo-variable \$\$ represents the value returned by the complete action. For example, the action:

```
{  $$ = 1;  }
```

returns the value of one; in fact, that is all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2, ... \$n. These refer to the values returned by components 1 through n of the right side of a rule, with the components numbered from left to right. If the rule is:

```
A      :  B  C  D  ;
```

then \$2 has the value returned by C, and \$3 the value returned by D. The rule:

```
expr   :  ' ( '  expr  ' ) '  ;
```

provides a common example. One would expect the value returned by this rule to be the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by:

```
expr   :  ' ( '  expr  ' ) '
        {
            $$ = $2 ;
        }
```

Basic Specifications

By default, the value of a rule is the value of the first element in it (\$1). Thus, grammar rules of the form:

```
A      :      B      ;
```

often need not have an explicit action. In previous examples, all the actions came at the end of rules. Sometimes it is desirable to get control before a rule is fully parsed. The **yacc** program permits an action to be written in the middle of a rule as well as at the end. This action is assumed to return a value accessible through the usual \$ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule below the effect is to set x to 1 and y to the value returned by C.

```
A      :      B
      {
          $$ = 1;
      }
      C
      {
          x = $2;
          y = $3;
      }
      ;
```

Actions that do not terminate a rule are handled by **yacc**, which manufactures a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the one triggered by recognizing this added rule. The **yacc** program treats the above example as if it had been written as follows (where **\$ACT** is an empty action):

```
$ACT   :      /* empty */
      {
          $$ = 1;
      }
      ;

A      :      B  $ACT  C
      {
          x = $2;
          y = $3;
      }
      ;
```

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired.

For example, suppose there is a C function node written so that the call:

```
node( L, n1, n2 )
```

creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly-created node. Then a parse tree can be built by supplying actions such as:

```
expr      :      expr '+' expr
          {
              $$ = node( '+', $1, $3 );
          }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks **%{** and **%}**. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
%{      int variable = 0;      %}
```

could be placed in the declarations section, making **variable** accessible to all of the actions. Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far, all the values are integers. A discussion of values of other types is found in the section "Advanced Topics."

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, that represents the kind of token being read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

The parser and the lexical analyzer must agree on these token numbers for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically.

Basic Specifications

For example, suppose that the token name `DIGIT` has been defined in the declarations section of the `yacc` specification file. The relevant portion of the lexical analyzer might look like:

```
int yylex()
{
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch (c)
    {
        ...
        case '0':
        case '1':
        ...
        case '9':
            yylval = c - '0';
            return (DIGIT);
        ...
    }
    ...
}
```

to return the appropriate token.

The intent is to return a token number of `DIGIT` and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the subroutines section of the specification file, the identifier `DIGIT` is defined as the token number associated with the token `DIGIT`.

This mechanism leads to clear, easily-modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names `if` or `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

In the default situation, token numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. If the `yacc` command is invoked with the `-d` option, a file called `y.tab.h` is generated. The `y.tab.h` file contains `#define` statements for the tokens.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker's token number must be 0 or negative. This token number cannot be redefined by the user. Thus, every lexical analyzer should be prepared to return 0 or a negative number as a token upon reaching the end of its input.

A very useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. The **lex** utility can easily be used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) that do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

Parser Operation

The **yacc** command turns the specification file into a C language procedure, which parses the input according to the specification given. The algorithm which is used to go from the specification to the parser is complex and will not be discussed here. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite-state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite-state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

4

The machine has only four actions available—**shift**, **reduce**, **accept**, and **error**. The parser does a step in the following manner:

1. Based on its current state, the parser decides whether it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto or popped off the stack, and in the look-ahead token being processed or left alone.

The shift action is the most common one the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56, there may be an action:

```
IF    shift 34
```

which says that if the look-ahead token is IF, then the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The **reduce** action keeps the stack from growing without bounds. The **reduce** actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right side with the left side. It may be necessary to consult the look-ahead token to decide whether or not to **reduce**. (Usually it is not necessary.) In fact, the default action (represented by a dot) is often a **reduce** action.

The **reduce** actions are associated with individual grammar rules. These rules are also given small integer numbers, and this leads to some confusion. The action:

```
.   reduce 18
```

refers to grammar rule 18, while the action:

```
IF   shift 34
```

refers to state 34.

Suppose the rule:

```
A   :   x y z   ;
```

is being reduced. The **reduce** action depends on the left-hand symbol (A, in this case) and the number of symbols on the right-hand side (three, in this case). To reduce, first pop three states off the top of the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer served any useful purpose. After these states have been popped, the state the parser was in before beginning to process the rule is uncovered. Using this uncovered state and the symbol on the left side of the rule, perform what is, in effect, a shift of A. A new state is obtained and pushed onto the stack, and parsing continues. However, there are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. In any case, the uncovered state contains an entry such as:

```
A   goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the **reduce** action turns back the clock in the parse, popping the states off the stack to go back to the state where the right side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off the stacks. The uncovered state is the current state.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, a parallel stack holds the values returned from the lexical analyzer and the actions. When a **shift** takes place, the external variable `yylval` is copied onto the value stack. After the return from the user code,

Parser Operation

the reduction is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudo-variables **\$1**, **\$2**, and so on, refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker, and indicates that the parser has done its job. The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider the following as a **yacc** specification:

```
4 %token DING DONG DELL
%%
rhyme : sound place
      ;
sound : DING DONG
      ;
place : DELL
      ;
```

When **yacc** is invoked with the **-v** option, a file called **y.output** is produced with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) follows:

```

state 0
    $accept : _rhyme $end
    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme_$end
    $end accept
    . error

state 2
    rhyme : sound_place
    DELL shift 5
    . error

    place goto 4

state 3
    sound : DING_DONG
    DONG shift 6
    . error

state 4
    rhyme : sound place_ (C)
    . reduce 1

state 5
    place : DELL_ (S)
    . reduce 3

state 6
    sound : DING DONG_ (S)
    . reduce 2
    
```

The actions for each state are specified, and there is a description of the parsing rules being processed in each state. The `_` character is used to indicate what has been seen and what is yet to come in each rule. The following input:

DING DONG DELL

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read

Parser Operation

and becomes the look-ahead token. The action in state 0 on DING is **shift 3**; state 3 is pushed onto the stack; and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**; state 6 is pushed onto the stack; and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by:

```
sound : DING DONG
```

which is rule 2. Two states, 6 and 3, are popped off the stack, uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**),

```
sound goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, DELL, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The **goto** in state 2 on **place** (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a **goto** on **rhyme**; causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained; indicated by **\$end** in the **y.output** file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, and so forth. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule:

$$\text{expr} : \text{expr} \text{ '-' } \text{expr}$$

is a natural way of expressing the fact that one way of forming an arithmetic expression is to join two other expressions with a minus sign. Unfortunately, this grammar rule does not completely specify how all complex inputs should be structured. For example, if the input is

$$\text{expr} - \text{expr} - \text{expr}$$

the rule allows this input to be structured as either:

$$(\text{expr} - \text{expr}) - \text{expr}$$

or:

$$\text{expr} - (\text{expr} - \text{expr})$$

(The first is called left association; the second, right association.)

The **yacc** program detects such ambiguities when it is attempting to build the parser. Given the input:

$$\text{expr} - \text{expr} - \text{expr}$$

consider the problem that confronts the parser. When the parser has read the second *expr*, the visible input:

$$\text{expr} - \text{expr}$$

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying it, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input:

$$- \text{expr}$$

and reduce again. The effect of this is to take the left associative interpretation.

Ambiguity and Conflicts

Alternatively, if the parser sees:

`expr - expr`

it could defer the immediate application of the rule and continue reading the input until it sees:

`expr - expr - expr`

It could then apply the rule to the rightmost three symbols, reducing them to *expr*, leaving:

`expr - expr`

Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read:

`expr - expr`

4

the parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a **shift-reduce** conflict. The parser may also have a choice of two legal reductions. This is called a **reduce-reduce** conflict. Note that there are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a disambiguating rule.

The **yacc** program invokes two default disambiguating rules:

1. In a **shift-reduce** conflict, the default is to do the shift.
2. In a **reduce-reduce** conflict, the default is to reduce by the earlier grammar rule (in the **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. This is why most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider:

```
stat      :  IF  '('  cond  ')'  stat
          |  IF  '('  cond  ')'  stat  ELSE  stat
          ;
```

which is a fragment from a programming language involving an **if-then-else** statement. In these rules, **IF** and **ELSE** are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple **if** rule and the second, the **if-else** rule.

These two rules form an ambiguous construction because input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways, either:

```
IF  (  C1  )
{
    IF  (  C2  )
        S1
}
ELSE
    S2
```

Ambiguity and Conflicts

or:

```
IF ( C1 )  
{  
    IF ( C2 )  
        S1  
    ELSE  
        S2  
}
```

where the second interpretation is the one given in most programming languages having this construct; each ELSE is associated with the last preceding un-ELSE'd IF. In this example, consider the situation where the parser has seen:

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ELSE. It can immediately reduce by the simple if rule to get:

```
IF ( C1 ) stat
```

and then read the remaining input:

```
ELSE S2
```

and reduce:

```
IF ( C1 ) stat ELSE S2
```

by the **if-else** rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted and S2 read; then the right-hand portion of:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get:

```
IF ( C1 ) stat
```

this can be reduced by the simple **if** rule, leading to the second of the above groupings of the input, which is usually desired.

Once again, the parser can do two valid things; there is a **shift-reduce** conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, **ELSE**, and particular inputs, such as:

```
IF ( C1 ) IF ( C2 ) S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. These are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE
state 23
    stat : IF ( cond ) stat_      (18)
    stat : IF ( cond ) stat_ELSE stat
ELSE      shift 45
.         reduce 18
```

where the first line describes the conflict; giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Remember that the underline marks the portion of the grammar rules, which has been seen. Thus, in the example, in state 23, the parser has seen input corresponding to:

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two things. If the input symbol is **ELSE**, it is possible to shift into state 45. State 45 will have, as part of its description, the line:

```
stat : IF ( cond ) stat ELSE_stat
```

because the **ELSE** will have been shifted in this state. In state 23, the alternative action (describing a dot, **.**) is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not **ELSE**, the parser reduces to:

```
stat : IF '(' cond ')'
```

by grammar rule 18.

Ambiguity and Conflicts

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed in parentheses after those rules, which can be reduced. In most states, a reduce action is possible in the state, and this is the default command. The user who encounters unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly-used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than those constructed from unambiguous grammars. The basic notion is to write grammar rules of the form:

```
expr : expr OP expr
```

and:

```
expr : UNARY expr
```

for all desired binary and unary operators. This creates a very ambiguous grammar with many parsing conflicts. To avoid ambiguity, the user specifies the precedence or binding strength of all the operators and the associates of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the precedences and associates.

The precedences and associativities are attached to tokens in the declarations section. This is done with a series of lines, beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left-associative and have lower precedence than star and slash, which are also left-associative. The keyword **%right** is used to describe right-associative operators, and the keyword **%nonassoc** is used to describe operators, like **.LT.** in FORTRAN, that may not associate with themselves.

Precedence

Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description:

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NAME
    ;
```

might be used to structure the input:

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( (c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus (-).

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword **%prec** changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal.

For example, the rules:

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
    ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a **reduce-reduce** conflict or a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used and the conflicts are reported.
4. If there is a **shift-reduce** conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (**shift** or **reduce**) associated with the higher precedence. If precedences are equal, then associativity is used. Left-associative implies **reduce**; right-associative implies **shift**; nonassociating implies **error**.

Precedence

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the precedence specification may disguise errors in the input grammar. It is a good idea to be sparing with precedence and use them in a cookbook fashion until you have experience. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse-tree storage, delete or alter symbol-table entries, and/or set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides the token name **error**. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token, and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form:

```
stat : error
```

means that, on a syntax error, the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, the parser may make a false start in the middle of a statement and end up reporting a second error where there is actually no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol-table space, and so on.

Error Handling

Error rules such as these are very general but difficult to control. Rules like:

```
stat : error ';' ;
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement to the next semicolon. Tokens following the error and preceding the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any cleanup action associated with it, performed.

Another form of **error** rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example is one way to do this:

```
input : error '\n'
      {
          (void) printf( "Reenter last line: " );
      }
      input
      {
          $$ = $4;
      }
      ;
```

There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. So, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement:

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as follows:

```
input : error '\n'
      {
          yyerrok;
          (void) printf( "Reenter last line: " );
      }
      input
      {
          $$ = $4;
      }
      ;
```

As previously mentioned, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement:

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by `yylex` would presumably be the first token in a legal statement. The old illegal token would have to be discarded and the **error** state reset. A rule similar to the following example could perform this:

```
stat      :   error
          {
            resynch();
            yyerrok ;
            yyclearin;
          }
          ;
```

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language subroutines, named *y.tab.c*. The function produced by **yacc** is called **yyparse()**; it is an integer-valued function. When **yyparse** is called, it repeatedly calls **yylex()**, the lexical analyzer supplied by the user to obtain input tokens. (See "Lexical Analysis.") Eventually, an error is detected, **yyparse()** returns the value 1, and no error recovery is possible; or else the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse()** returns the value 0.

The user must provide a certain amount of environment for this parser to obtain a working program. For example, as with every C language program, a routine called **main()** must be defined that eventually calls **yyparse()**. In addition, a routine called **yyerror()** is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of **main()** and **yyerror()**. The library is accessed by a **-ly** argument to the **cc(CP)** command or the loader. The source codes:

```
main()
{
    return (yyparse());
}
```

and

```
# include <stdio.h>

yyerror(s)
    char *s;
{
    (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs.

The argument to **yerror()** is a string containing an error message, usually the string **syntax error**. The average application wants to do better than this. Ordinarily, the program should keep track of the input-line number, and print it along with the message when a syntax error is detected. The external integer variable **yychar** contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main()** routine is probably supplied by the user (to read arguments, and such.), the **yacc** library is useful only in small projects or the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using a debugger (**adb**, **sdb**, or **codeview**).

Hints for Preparing Specifications

This part contains miscellaneous hints on preparing efficient, easily changeable, and clear specifications. The individual subsections are largely independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. Here are a few style hints:

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This helps in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. Put all rules with the same left side together. Put the left side in only once and let each successive rule begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be added easily.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 is written in this style, as are other examples in this section (where space permits). The user must decide about these stylistic questions. However, the central problem is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so-called left-recursive grammar rules. Rules of the form:

```
name      :   name  rest_of_rule  ;
```

match this algorithm. These rules, such as:

```
list      :   item
          |   list  ','  item
          ;
```

and:

```
seq       :   item
          |   seq  item
          ;
```

frequently arise during the writing of specifications for sequences and lists. In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq       :   item
          |   item  seq
          ;
```

the parser is a bit bigger, and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning; if so, consider writing the sequence specification as:

```
seq       :   /* empty */
          |   seq  item
          ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it has not seen enough to know.

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might normally want to delete blanks, but not within quoted strings; or names might be entered into a symbol table in declarations; but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example:

```
%{
    int dflag;
}%
... other declarations ...

%%

prog  :   decls  stats
      ;

decls :   /* empty */
      {
          dflag = 1;
      }
      |   decls  declaration
      ;

stats :   /* empty */
      {
          dflag = 0;
      }
      |   stats  statement
      ;

... other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag **dflag** is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back-door approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass to the lexical analyzer information telling it one instance of **if** is a keyword and another instance is a variable. The user can make a stab at it using the mechanism described in the last subsection, but it is not easy.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, that is, forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced Topics

This part discusses a number of advanced features of **yacc**.

Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The **YYACCEPT** macro causes **yyparse()** to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; **yyerror()** is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context-sensitive syntax checking.

4

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is the same as with ordinary actions: a dollar sign followed by a digit.

```

sent      :   adj  noun  verb  adj  noun
          {
              look at the sentence ...
          }
          ;

adj       :   THE
          {
              $$ = THE;
          }
          |   YOUNG
          {
              $$ = YOUNG;
          }
          ;

```



```

noun      :      DOG
          {
            $$ = DOG;
          }
          |      CRONE
          {
            if( $0 == YOUNG )
            {
              (void) printf( "what?\n" );
            }
            $$ = CRONE;
          }
          ;
          ...

```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made to ascertain that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times, this mechanism prevents a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

4

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The **yacc** program can also support values of other types, including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type-checked. **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with each token and nonterminal symbol having a value. When the value is referenced through a **\$\$** or **\$n** construction, **yacc** will automatically insert the appropriate union name so that no unwanted conversions take place. In addition, type-checking commands such as **lint** are far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user; since other subroutines, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values **yacc** cannot easily determine the type.

Advanced Topics

To declare the union, the user includes

```
%union
{
    body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables **yylval** and **yyval** to have type equal to this union. If **yacc** was invoked with the **-d** option, the union declaration is copied onto the **y.tab.h** file as **YYSTYPE**.

Once **YYSTYPE** is defined, the union member names must be associated with the various terminal and nonterminal names. The construction:

```
<name>
```

4

is used to indicate a union member name. If this follows one of the keywords, **%token**, **%left**, **%right**, or **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying:

```
%left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say:

```
%type <nodetype> expr stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**.

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left-context values (such as `$0`) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between `<` and `>` immediately after the first `$`. The example:

```
rule    :    aaa
        {
            $<intval>$ = 3;
        }
        bbb
    {
        fun( $<intval>2, $<other>0 );
    }
    ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. The facilities in this subsection are not triggered until they are used. In particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of `$n` or `$$` to refer to something with no defined type is diagnosed. If these facilities are not triggered, the `yacc` value stack is used to hold ints.

yacc Input Syntax

This section has a description of the `yacc` input syntax as a `yacc` specification. Context dependencies and the like are not considered. Ironically, although `yacc` accepts an LALR(C) grammar, the `yacc` input-specification language is most naturally specified as an LR(S) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, then it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (such as skipping blanks, newlines, and comments) is a colon. If so, it returns the token `C_IDENTIFIER`. Otherwise, it returns `IDENTIFIER`. Literals (quoted strings) are also returned as `IDENTIFIERS`, but never as part of `C_IDENTIFIER`s.

Advanced Topics

```
/* grammar for the input to yacc */

/* basic entries */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by a : */
%token NUMBER /* [0-9]+ */

/* reserved words: %type=>TYPE %left=>LEFT, etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the %{ mark */
%token RCURL /* the %} mark */

/* ASCII character literals stand for themselves */
%token spec

%%

spec : defs MARK rules tail
;
tail : MARK
{
    In this action, eat up the rest of the file
}
| /* empty: the second MARK is optional */
;

defs : /* empty */
| defs def
;
def : START IDENTIFIER
| UNION
{
    Copy union definition to output
}
| LCURL
{
    Copy C code to output file
}
| RCURL
| rword tag nlist
;

rword : TOKEN
| LEFT
| RIGHT
| NONASSOC
| TYPE
;

tag : /* empty: union tag is optional */
| '<' IDENTIFIER '>'
;

nlist : nmno
| nlist nmno
| nlist ',' nmno
;

nmno : IDENTIFIER /* Note: literal illegal with % type */
| IDENTIFIER NUMBER /* Note: illegal with % type */
;
```

```
/* rule section */

rules : C_IDENTIFIER rbody prec
      | rules rule
      ;

rule  : C_IDENTIFIER rbody prec
      | '[' rbody prec
      ;

rbody : /* empty */
      | rbody IDENTIFIER
      | rbody act
      ;

act   : '{'
      {
          Copy action translate $$ etc.
      }
      ;

prec  : /* empty */
      | PREC IDENTIFIER
      | PREC IDENTIFIER act
      | prec ';'
      ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers, labeled **a** through **z**, and accepts arithmetic expressions made up of the operators:

```
+, -, *, /, % (mod operator), & (bitwise and),  
| (bitwise or), and assignments.
```

4 If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator shows how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately; line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.


```

%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */
%% /* beginning of rules section */

list      : /* empty */
           | list stat '\n'
           | list error '\n'
           {
             yyerrok;
           }
           ;

stat       : expr
           {
             (void) printf( "%d\n", $1 );
           }
           | LETTER '=' expr
           {
             regs[$1] = $3;
           }
           ;

expr       : '(' expr ')'
           {
             $$ = $2;
           }
           | expr '+' expr
           {
             $$ = $1 + $3;
           }
           | expr '-' expr
           {
             $$ = $1 - $3;
           }
           | expr '*' expr
           {
             $$ = $1 * $3;
           }
           | expr '/' expr
           {
             $$ = $1 / $3;
           }
           | exp '%' expr
           {
             $$ = $1 % $3;
           }
           | expr '&' expr
           {
             $$ = $1 & $3;
           }

```

Examples

```

    }
    |   expr '|' expr
    {
        $$ = $1 | $3;
    }
    |   '-' expr %prec UMINUS
    {
        $$ = -$2;
    }
    |   LETTER
    {
        $$ = reg[$1];
    }
    |   number
    ;

number
:   DIGIT
{
    $$ = $1; base = ($1==0) ? 8 ; 10;
}
|   number DIGIT
{
    $$ = base * $1 + $2;
}
;

%%      /* beginning of subroutines section */

int yylex( ) /* lexical analysis routine */
{
    /* return LETTER for lowercase letter, */
    /* yylval = 0 through 25 */
    /* returns DIGIT for digit, yylval = 0 through 9 */
    /* all other characters are returned immediately */
    int c;
        /*skip blanks*/
    while ((c = getchar()) == ' ')
        ;

        /* c is now nonblank */

    if (islower(c))
    {
        yylval = c - 'a';
        return (LETTER);
    }
    if (isdigit(c))
    {
        yylval = c - '0';
        return (DIGIT);
    }
    return (c);
}

```

2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example (Example 1) is modified to provide a desk calculator that does floating-point interval arithmetic. The calculator understands floating-point constants; the arithmetic operations $+$, $-$, $*$, $/$, and unary $-$ a through z . Moreover, it also understands intervals written:

(X, Y)

where X is less than or equal to Y . There are 26 interval-valued variables, A through Z , that may also be used. The usage is similar to that in Example 1; assignments return no value and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, **INTERVAL**, by using **typedef**. The **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of **YYERROR** to handle error conditions; this is division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines; these are:

$2.5 + (3.5 - 4.)$

and:

$2.5 + (3.5, 4)$

Examples

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator—one when the left operand is a scalar, and one rule when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued, until they are forced to become intervals.

4 This way of handling multiple types is very instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts, even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating-point constants. The C language library routine `atof()` is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser and thence error recovery.

```

%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
    double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST      /* floating point constant */

%type <dval> dexp        /* expression */

%type <vval> vexp        /* interval expression */

/* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS /* precedence for unary minus */

%% /* beginning of rules section */

lines : /* empty */
      | lines line
      ;
line  : dexp '\n'
      {
          (void) printf("%15.8f\n", $1);
      }
      | vexp '\n'
      {
          (void) printf("%15.8f, %15.8f\n", $1.lo, $1.hi);
      }
      | DREG '=' dexp '\n'
      {
          dreg[$1] = $3;
      }
      | VREG '=' vexp '\n'
      {
          vreg[$1] = $3;
      }

```

Examples

```

| error '\n'
| {
|     yyerrok;
| }
;

dexp : CONST
| DREG
| {
|     $$ = dreg[$1];
| }
| dexp '+' dexp
| {
|     $$ = $1 + $3;
| }
| dexp '-' dexp
| {
|     $$ = $1 - $3;
| }
| dexp '*' dexp
| {
|     $$ = $1 * $3;
| }
| dexp '/' dexp
| {
|     $$ = $1 / $3;
| }
| '-' dexp %prec UMINUS
| {
|     $$ = -$2;
| }
| '(' dexp ')'
| {
|     $$ = $2;
| }
;

vexp : dexp
| {
|     $$hi = $$lo = $1;
| }
| '(' dexp ',' dexp ')'
| {
|     $$lo = $2;
|     $$hi = $4;
|     if( $$lo > $$hi )
|     {
|         (void) printf("interval out of order \n");
|         YYERROR;
|     }
| }
| VREG
| {
|     $$ = vreg[$1];
| }
| vexp '+' vexp
| {
|     $$hi = $1hi + $3hi;
|     $$lo = $1lo + $3lo;
| }
| dexp '+' vexp
| {
|     $$hi = $1 + $3hi;
|     $$lo = $1 + $3lo;
| }

```



```

| vexp '-' vexp
{
    $$hi = $1hi - $3lo;
    $$lo = $1lo - $3hi;
}
| dexp '-' vdep
{
    $$hi = $1 - $3lo;
    $$lo = $1 - $3hi
}
| vexp '*' vexp
{
    $$ = vmul( $1lo,$hi,$3 )
}
| dexp '*' vexp
{
    $$ = vmul( $1, $1, $3 )
}
| vexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1lo, $1hi, $3 )
}
| dexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1lo, $1hi, $3 )
}
| '-' vexp %prec UMINUS
{
    $$hi = -$2lo; $$lo = -$2hi
}
| '(' vexp ')'
{
    $$ = $2
}
;

%%      /* beginning of subroutines section */

# define BSZ 50      /* buffer size for floating point number */

/* lexical analysis */

int yylex( )
{
    register int c;

                                /* skip over blanks */
    while ((c = getchar()) == ' ')
    ;
    if (isupper(c))
    {
        yylval.ival = c - 'A'
        return (VREG);
    }
    if (islower(c))
    {
        yylval.ival = c - 'a',
        return( DREG );
    }

    /* gobble up digits. points, exponents */
    if (isdigit(c) || c == '.')

```

Examples

```
{
    char buf[BSZ+1], *cp = buf;
    int dot = 0, exp = 0;

    for(;; (cp - buf) < BSZ ; ++cp, c = getchar())
    {
        *cp = c;
        if (isdigit(c))
            continue;
        if (c == '.')
        {
            if (dot++ || exp)
                return ('.'); /* will cause syntax error */
            continue;
        }
        if (c == 'e')
        {
            if (exp++)
                return ('e'); /* will cause syntax error */
            continue;
        }
        /* end of number */
        break;
    }

    *cp = '\0';
    if (cp - buf >= BSZ)
        (void) printf("constant too long - truncated\n");
    else
        ungetc(c, stdin); /* push back last char read */
    yylval.dval = atof(buf);
    return (CONST);
}
return (c);
}
INTERVAL
hilo(a, b, c, d)
double a, b, c, d;
{
    /* returns the smallest interval containing a, b, c, and d */

    /* used by *,/ routine */
    INTERVAL v;

    if (a > b)
    {
        v.hi = a;
        v.lo = b;
    }
    else
    {
        v.hi = b;
        v.lo = a;
    }
    if (c > d)
    {
        if (c > v.hi)
            v.hi = c;
        if (d < v.lo)
            v.lo = d;
    }
    else
    {
        if (d > v.hi)
            v.hi = d;
    }
}
```

```

        if (c < v.lo)
            v.lo = c;
    }
    return (v);
}
INTERVAL
vmul(a, b, v)
    double a, b;
    INTERVAL v;
{
    return (hilo(a * v.hi, a * v.lo, b * v.hi, b * v.lo));
}
dcheck(v)
    INTERVAL v;
{
    if (v.hi >= 0. && v.lo <= 0.)
    {
        (void) printf("divisor interval contains 0.\n");
        return (C);
    }
    return (0);
}
INTERVAL
vdiv(a, b, v)
    double a, b;
    INTERVAL v;
{
    return (hilo(a / v.hi, a / v.lo, b / v.hi, b / v.lo));
}

```


Chapter 5

make

Introduction 5-1

Basic Features 5-2

Description Files and Substitutions 5-6

Comments 5-6

Continuation Lines 5-6

Macro Definitions 5-6

General Form 5-7

Dependency Information 5-7

Output Translations 5-7

Recursive **Makefiles** 5-9

Suffixes and Transformation Rules 5-9

Implicit Rules 5-10

Executable Commands 5-12

Extensions of \$*, \$@, and \$< 5-13

Archive Libraries 5-13

The Tilde in SCCS File Names 5-16

The Null Suffix 5-18

include Files 5-18

SCCS Makefiles 5-19

Dynamic Dependency Parameters 5-19

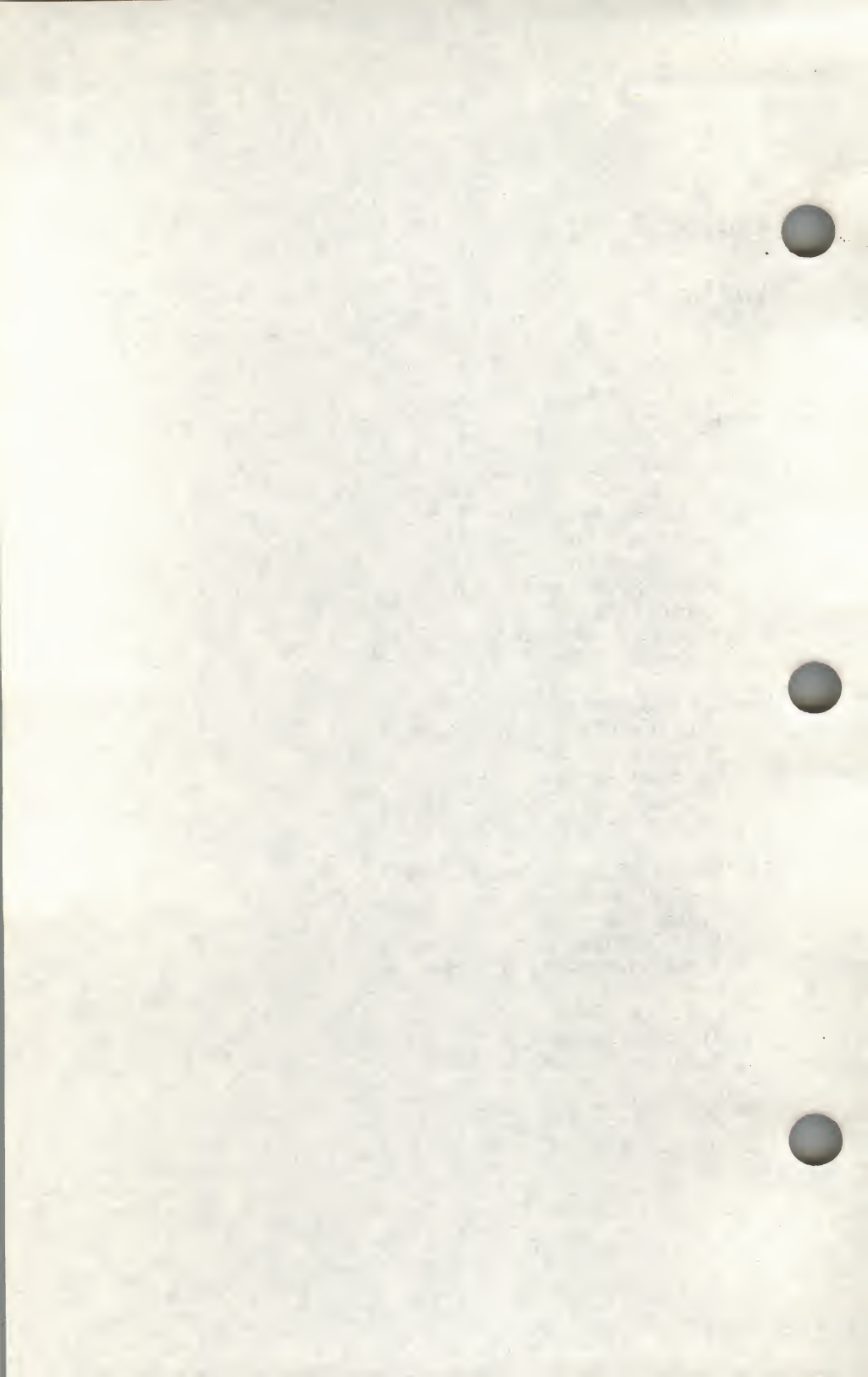
Command Usage 5-20

The **make** Command 5-20

Environment Variables 5-22

Suggestions and Warnings 5-24

Internal Rules 5-25



Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

The **make**(CP) command, documented in the *Programmer's Reference*, provides a method for maintaining up-to-date versions of programs that consist of files that may be generated in a variety of ways.

An individual programmer can easily forget:

- file-to-file dependencies
- files that were modified and the impact that has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationships among files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to:

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up-to-date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile** or **Makefile**. If this file is under SCCS control, its name is **s.makefile** or **s.Makefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target, regardless of the number of files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up-to-date. The target file is regenerated if it has not been modified since the dependents' modification. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time when a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- file names and times last modified from the filesystem
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **math** library. By convention, the output of the C language compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs.h*, but that *z.c* does not. That is, *x.c* and *y.c* have the line:

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog : x.o y.o z.o
      cc x.o y.o z.o -lm -o prog

x.o y.o : defs.h
```

If this information were stored in a file named *makefile*, the command

make

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs.h*. In the example above, the first line states that **prog** depends on three *.o* files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that *x.o* and *y.o* depend on the file *defs.h*. From the file system, **make** discovers that there are three *.c* files corresponding to the needed *.o* files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc -c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```

prog : x.o y.o z.o
      cc x.o y.o z.o -lm -o prog
x.o : x.c defs.h
      cc -c x.c
y.o : y.c defs.h
      cc -c y.c
z.o : z.c
      cc -c z.c

```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the *defs.h* file has been edited, *x.c* and *y.c* (but not *z.c*) are recompiled; and then **prog** is created from the new *x.o* and *y.o* files and the existing *z.o* file. If only the file *y.c* had changed, only it is recompiled, but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command:

make x.o

would regenerate *x.o* if *x.c* or *defs.h* had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce files with those names. These entries can take advantage of **make**'s ability to generate files and substitute macros. (For information about macros, see "Description Files and Substitutions" further along in this chapter.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Basic Features

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equal sign, followed by what the macro stands for. A macro is invoked by preceding the name with a dollar sign. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$ (CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

\$\$, **\$@**, **\$\$?**, and **\$\$<** are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under “Description Files and Substitutions.”) The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
      cc $(OBJECTS)  $(LIBES)  -o prog
. . .
```

The command:

```
make LIBES="-ll -lm"
```

loads the three objects with both the **lex** (**-ll**) and the **math** (**-lm**) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX System commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. An example of the description file follows:

```

# Description file for the make command
FILES = Makefile defs.h main.c doname.c misc.c
      files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
      dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make: $(OBJECTS)
      $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
      @size make

$(OBJECTS): defs.h

cleanup:
      -rm *.o gram.c
      -du

install:
      @size make /usr/bin/make
      cp make /usr/bin/make && rm make

lint : dosys.c doname.c files.c main.c misc.c gram.c
      $(LINT) dosys.c doname.c files.c main.c misc.c \
      gram.c
      # print files that are out-of-date
      # with respect to "print" file.

print: $(FILES)
      pr $? | $(LP)
      touch print

```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```

cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc main.o doname.o misc.o files.o dosys.o
   gram.o -lld -o make
13188 + 3348 + 3044 = 19580

```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign (@) in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a sharp (#) and all characters on the same line after it are ignored. Blank lines and lines beginning with sharps are totally ignored.

Continuation Lines

If a noncomment line is too long, it can be continued by using a backslash (\). If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

5

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make's** own rules. (See Figure 5-2 at the end of the chapter.)

General Form

A dependency line has the form:

```
target1 [target2 ...] [:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
. . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as `*` and `?` are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with tabs immediately following a dependency line. A command is any string of characters not including a sharp (`#`) except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single- or a double-colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line per target name. If the target is out-of-date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out-of-date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the “Archive Libraries” section, later in this chapter.)

5

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of `$(macro)` is evaluated. For each appearance of `string1` in the evaluated macro, `string2` is substituted. The meaning of finding `string1` in `$(macro)` is that the evaluated `$(macro)` is considered as a series of strings, each delimited by white space (blanks or tabs). Thus,

Description Files and Substitutions

the occurrence of **string1** in **\$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. This type of translation is useful when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (which are those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB) (a.o) $(LIB) (b.o) $(LIB) (c.o)
$(CC) -c $(CFLAGS) $(?:.o=.c)
$(AR) $(ARFLAGS) $(LIB) $?
rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. For testing purposes, **make -n ...** can be executed and everything that would have been done will be printed, including output from lower-level invocations of **make**. If the sequence **\$(MAKE)** appears anywhere in a shell-command line, the line is executed even if the **-n** flag is set. Since the **-n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile(s)** describes a set of software subsystems.

Suffixes and Transformation Rules

The **make** program uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **-r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name **.SUFFIXES**. The **make** program searches for a file with any of the suffixes on the list. If one is found, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. Thus, the name of the rule to transform a **.r** file to a **.o** file is **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **\$*** is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro **\$<** is the full name of the dependent that caused the action.

The order of the suffix list is significant because the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for **.SUFFIXES** in the description file. The dependents are added to the usual list. A **.SUFFIXES** line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

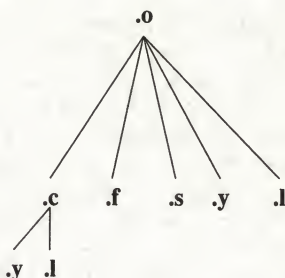
Implicit Rules

The **make** program uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	object file
.c	C source file
.c~	SCCS C source file
.f	FORTTRAN source file
.f~	SCCS FORTRAN source file
.s	assembler source file
.s~	SCCS Assembler source file
.y	yacc source grammar
.y~	SCCS yacc source grammar
.l	lex source grammar
.l~	SCCS ex source grammar
.h	header file
.h~	SCCS header file
.sh	shell file
.sh~	SCCS shell file

Figure 5-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

Figure 5-1 Summary of Default Transformation Path



If the file *x.o* is needed and an *x.c* is found in the description or directory, the *x.o* file would be compiled. If there is also an *x.l*, that source file would be run through *lex* before compiling the result. However, if there is no *x.c* but there is an *x.l*, **make** would discard the intermediate C language file and use the direct link as shown in Figure 5-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command:

```
make CC=newcc
```

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus:

```
make "CFLAGS=-g"
```

causes the **cc** command to include debugging information.

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell, after substituting for macros. The printing is suppressed in the silent mode (**-s** option of the **make** command) or if the command line in the description file begins with an **@** sign. The **make** program normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if: the **-i** flag has been specified on the **make** command line, the fake target name **.IGNORE** appears in the description file, or the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (like **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before any command is issued, certain internally-maintained macros are set. The **\$@** macro is set to the full target name of the current target. The **\$@** macro is evaluated only for explicitly named dependencies. The **\$?** macro is set to the string of names that were found to be younger than the target. The **\$?** macro is evaluated along with explicit rules from the **makefile**. If the command was generated by an implicit rule, the **\$<** macro is the name of the related file that caused the action, and the **\$*** macro is the prefix shared by the current and the dependent file names. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the target name **.DEFAULT** are used.

If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **\$(@D)**, **\$(@F)**, **\$(*D)**, **\$(*F)**, **\$(<D)**, and **\$(<F)**. (See below.)

Extensions of \$*, \$@, and \$<

The internally-generated macros \$*, \$@, and \$< are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: \$(@D), \$(@F), \$(*D), \$(*F), \$(<D), and \$(<F). The **D** refers to the directory part of the single-character macro. The **F** refers to the file name part of the single-character macro. These additions are useful when building hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell.

Thus, a command can be:

```
cd $(<D); $(MAKE) $(<F)
```

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library:

```
projlib(object.o)
```

or:

```
projlib( (encrypt) )
```

The second method actually refers to an entry point of an object file within the library. (The **make** program looks through the library, locates the entry point, and translates it to the correct object-file name.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib:: projlib(pfile1.o)
          $(CC) -c -O pfile1.c
          $(AR) $(ARFLAGS) projlib pfile1.o
          rm pfile1.o
projlib:: projlib(pfile2.o)
          $(CC) -c -O pfile2.c
          $(AR) $(ARFLAGS) projlib pfile2.o
          rm pfile2.o
```

and so on for each object.

Recursive Makefiles

This is tedious and error-prone. In most cases, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time.

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde (~) syntax will be described shortly.) The user may define other needed rules in the description file.

The two-member library mentioned earlier is then maintained with the following shorter **makefile**:

```
projlib:      projlib(pfile1.o) projlib(pfile2.o)
              @echo projlib up-to-date
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
    $(CC) -c $(CFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.o
```

Thus, the **\$@** macro is the **.a** target (**projlib**); the **\$<** and **\$*** macros are set to the out-of-date C language file, and the file name minus the suffix, respectively (**pfile1.c** and **pfile1**). The **\$<** macro (in the preceding rule) could have been changed to **\$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction:

```
projlib:      projlib(pfile1.o)
              @echo projlib up-to-date
```

Assume the object in the library is out-of-date with respect to *pfile1.c*. Also, there is no *pfile1.o* file.

1. **make projlib.**
2. Before **makeing projlib**, check each dependent of **projlib**.
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.

4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)
5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.
6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **\$@** (**=projlib**) and **\$*** (**=pfile1**).
7. Look for a rule **.X.a** and a file **\$.X**. The first **X** (in the **.SUFFIXES** list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **\$<** to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.
8. The library has been updated. Execute the command associated with the **projlib:** dependency. That is:

```
@echo projlib up-to-date
```

It should be noted that to let *pfile1.o* have dependencies, the following syntax is required:

```
projlib(pfile1.o): $(INCDIR)/stdio.h pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **\$%** macro is evaluated each time **\$@** is evaluated. If there is no current archive member, **\$%** is null. If an archive member exists, then **\$%** evaluates to the expression between the parentheses.

The Tilde in SCCS File Names

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX Operating System machines, this is acceptable because nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the file name part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde (~) is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is:

```
.c~.o:
$(GET) $(GFLAGS) $<
$(CC) $(CFLAGS) -c $*.c
-rm -f $*.c
```

5

Thus, the tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

.c~
.f~
.y~
.l~
.s~
.sh~
.h~
.c~:
.f~:
.sh~:
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS file name format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. The **make** program handles this case with the null suffix rule. Thus, to maintain the UNIX System program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
    $(CC) $(CFLAGS) $< -o $@
```

In fact, this **.c:** rule is internally defined, so no **makefile** is necessary at all. The user only needs to type:

```
make cat dd echo date
```

(all of which are UNIX System single-file programs), and all four C language source files are passed through the above shell command line associated with the **.c:** rule. The internally-defined single-suffix rules are:

```
.c:
.c~:
.f:
.f~:
.sh:
.sh~:
```

Others may be added to the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a file name, which the current invocation of **make** will read. Macros may be used in file names. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **includes** are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named *s.makefile* or *s.Makefile* exists, **make** will do a get on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **\$\$@** refers to the current “thing” to the left of the colon (which is **\$\$@**). Also the form **\$\$(@F)** exists, allowing access to the file part of **\$\$@**. Thus, in the following example:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX System software command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown
```

```
$(CMDS):    $$@.c
$(CC) -o $? -o $@
```

Obviously, this is a subset of all the single-file programs. For multiple-file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful form of the dependency parameter is **\$\$(@F)**. It represents the file name part of **\$\$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like this:

```
INCDIR = /usr/include

INCLUDES = \
    $(INCDIR)/stdio.h \
    $(INCDIR)/pwd.h \
    $(INCDIR)/dir.h \
    $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
    cp $? $@
    chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make(CP)** in the *Programmer's Reference*.

The make Command

The **make** command takes macro definitions, options, description file names, and target file names as arguments in the form:

```
make [ options ] [ macro definitions ] [ targets ]
```

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

- i ignores error codes returned by commands invoked. This mode is entered if the fake target name **.IGNORE** appears in the description file.
- s is silent mode. It does not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.
- r Does not use the built-in rules.
- n is no execute mode. It prints commands, but does not execute them. Even lines beginning with an **@** sign are printed.
- t touches the target files (causing them to be up-to-date) rather than issuing the usual commands.
- q is a question. The **make** command returns a zero or nonzero status code, depending on whether the target file is or is not up-to-date.

- p prints out the complete set of macro definitions and target descriptions.
- k abandons work on the current entry if something goes wrong, but continues on other branches that do not depend on the current entry.
- e are environment variables that override assignments within **makefiles**.
- f is a description file name. The next argument is assumed to be the name of a description file. A file name of - denotes the standard input. If there are no -f arguments, the file named *makefile*, or *Makefile*, or *s.[mM]akefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following arguments are evaluated in the same manner as flags:

- | | |
|-----------|---|
| .ECFAULT | If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists. |
| .IGNORE | Same as -i option. Ignores error codes returned by commands invoked. |
| .PRECIOUS | Dependents on this target are not removed when quit or interrupt is pressed. |
| .SILENT | Same as -s option. Uses silent mode and does not print command lines before executing. |

Finally, the remaining arguments are assumed to be the names of targets to be made, and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, **MAKEFLAGS**, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command-line flags and assignments in the **makefile** update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, the **MAKEFLAGS** macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **-f**, **-p**, and **-r** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).
4. Read the **makefile(s)**. The assignments in the **makefile(s)** override the environment. This order is chosen so that, when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **-e** flag is used. The **-e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make -e ...** is typed, the variables in the environment override the definitions in the **makefile**. Also **MAKEFLAGS** override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile(s)**
4. command line

The **-e** flag has the effect of rearranging the order to

1. internal definitions
2. **makefile(s)**
3. environment
4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has a:

```
#include "defs.h"
```

line, then the object file *x.o* depends on *defs.h*; the source file *x.c* does not. If *defs.h* is changed, nothing is done to the file *x.c* while file *x.o* must be recreated.

To discover what **make** would do, the **-n** option is very useful. The command:

```
make -n
```

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (such as adding a comment to an *include* file), the **-t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command:

```
make -ts
```

(with *ts* standing for touch silently) causes the relevant files to appear up-to-date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** is reproduced below.

Figure 5-2 make Internal Rules (Sheet 1 of 5)

```
#
#      SUFFIXES RECOGNIZED BY MAKE
#
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~
#
#      PREDEFINED MACROS
#
MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-r-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 5-2 make Internal Rules (Sheet 2 of 5)

```
#
# SINGLE SUFFIX RULES
#

.c:
    $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@

.c~:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
    -rm -f $*.c

.f:
    $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@

.f~:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
    -rm -f $*.f

.sh:
    cp $< $@; chmod 0777 $@

.sh~:
    $(GET) $(GFLAGS) $<
    cp $*.sh $*; chmod 0777 $@
    -rm -f $*.sh
```

Figure 5-2 make Internal Rules (Sheet 3 of 5)

```

#
# DOUBLE SUFFIX RULES
#
.c~.c .f~.f .s~.s .sh~.sh .y~.y .l~.l .h~.h:
$(GET) $(GFLAGS) $<

.c.a:
$(CC) -c $(CFLAGS) $<
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.o

.c~.a:
$(GET) $(GFLAGS) $<
$(CC) -c $(CFLAGS) $*.c
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.[co]

.c.o:
$(CC) $(CFLAGS) -c $<

.c~.o:
$(GET) $(GFLAGS) $<
$(CC) $(CFLAGS) -c $*.c
-rm -f $*.c

.f.a:
$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
$(AR) $(ARFLAGS) $@ $*.o
-rm -f $*.o

.f~.a:
$(GET) $(GFLAGS) $<
$(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
$(AR) $(ARFLAGS) $@ $*.o
-rm -f $*.[fo]

```


Figure 5-2 make Internal Rules (Sheet 4 of 5)

```
.f.o:
    $(F77) $(F77FLAGS) $(LDFLAGS) -c $.f

.f~.o:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) $(LDFLAGS) -c $.f
    -rm -f $.f

.s~.a:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $.o $.s
    $(AR) $(ARFLAGS) $@ $.o
    -rm -f $.[so]

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $.o $.s
    -rm -f $.s

.l.c :
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@

.l~.c:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $.l
    mv lex.yy.c $@
```

Figure 5-2 make Internal Rules (Sheet 5 of 5)

```

.l.o:
$(LEX) $(LFLAGS) $<
$(CC) $(CFLAGS) -c lex.yy.c
rm lex.yy.c
mv lex.yy.o $@
-rm -f $*.l

.l~.o:
$(GET) $(GFLAGS) $<
$(LEX) $(LFLAGS) $*.l
$(CC) $(CFLAGS) -c lex.yy.c
rm -f lex.yy.c $*.l
mv lex.yy.o $*.o

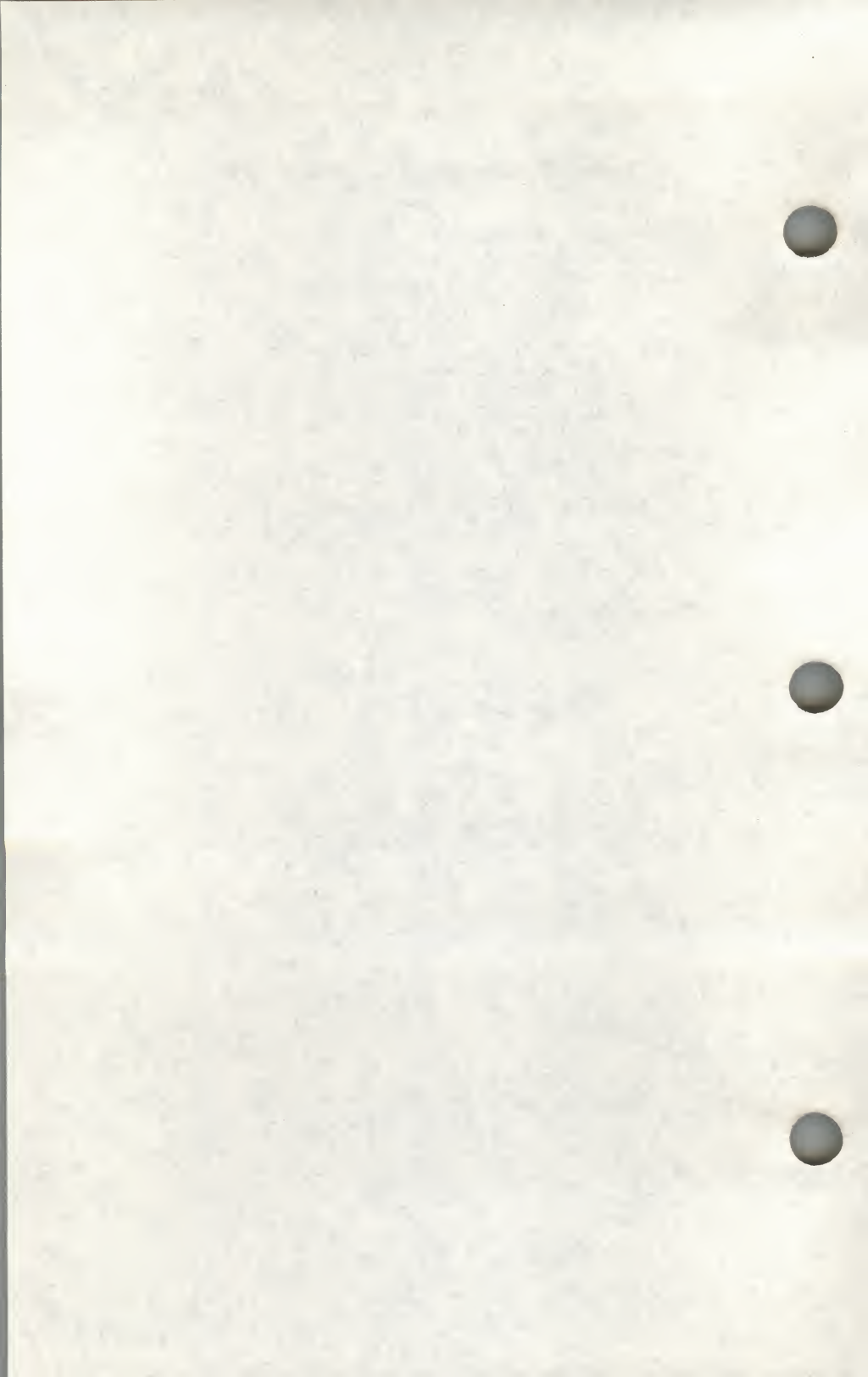
.y.c :
$(YACC) $(YFLAGS) $<
mv y.tab.c $@

.y~.c :
$(GET) $(GFLAGS) $<
$(YACC) $(YFLAGS) $*.y
mv y.tab.c $*.c
-rm -f $*.y

.y.o:
$(YACC) $(YFLAGS) $<
$(CC) $(CFLAGS) -c y.tab.c
rm y.tab.c
mv y.tab.o $@

.y~.o:
$(GET) $(GFLAGS) $<
$(YACC) $(YFLAGS) $*.y
$(CC) $(CFLAGS) -c y.tab.c
rm -f y.tab.c $*.y
mv y.tab.o $*.o

```



Chapter 6

The Link Editor

The Link Editor 6-1

Memory Configuration 6-1

Sections 6-2

Addresses 6-2

Binding 6-2

Object File 6-3

Link Editor Command Language 6-4

Expressions 6-4

Assignment Statements 6-5

Specifying a Memory Configuration 6-7

Section Definition Directives 6-8

Notes and Special Considerations 6-20

Changing the Entry Point 6-20

Use of Archive Libraries 6-20

Dealing With Holes in Physical Memory 6-23

Allocation Algorithm 6-24

Incremental Link Editing 6-24

DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections 6-26

Output File Blocking 6-27

Nonrelocatable Input Files 6-28

Syntax Diagram for Input Directives 6-30

The Link Editor

The commands **cc** and **ld** referred to in this section are documented in the *Programmer's Reference*. This chapter contains information on the Link Editor Command Language. The command language enables you to

- specify the memory configuration of the target machine
- combine the sections of an object file in arrangements other than the default
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and where they are located in memory. When you do need to be very precise in controlling the link editor output, you do it by means of the command language.

Link editor command language directives are passed in a file named on the **ld**(CP) command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

6

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K-1 are then not configured. Unconfigured memory is treated as

reserved or unusable by **ld**(CP). Nothing can ever be linked into unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or non-existent with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc., are with respect to the configured sections of the address space.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld**(CP) command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by **ld**(CP). **ld**(CP) accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld**(CP) can also be absolute files.

Files produced from the compilation system may contain, among others, sections called **.text** and **.data**. The **.text** section contains the instruction text (executable instructions); **.data** contains initialized data variables. For example, if a C program contained the global (i.e., not inside a function) declaration

```
int i = 100;
```

and the assignment

```
i = 0;
```

then compiled code from the C assignment is stored in **.text**, and the variable **i** is located in **.data**.

Link Editor Command Language

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 6-2, "Syntax Diagram for Input Directives.") Constants are as in C with a number recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, `_`. Symbols within an expression have the value of the address of the symbol only. `ld(CP)` does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

`ld(CP)` uses a `lex`-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

ADDR	BLOCK	GROUP	NEXT	RANGE	SPARE
ALIGN	COMMON	INFO	NOLOAD	REGIONS	PHY
ASSIGN	COPY	LENGTH	ORIGIN	SECTIONS	TV
BIND	DSECT	MEMORY	OVERLAY	SIZEOF	
addr	block	length	origin	sizeof	
align	group	next	phy	spare	
assign	l	o	range		
bind	len	org	s		

The operators that are supported, in order of precedence from high to low, are shown in Figure 6-1:

Figure 6-1 Operator Symbols

symbol
! ~ - (UNARY Minus)
* / %
+ - (BINARY Minus)
>> <<
== != > < <= >=
&
&&
= += -= *= /=

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses by means of the assignment statement. The syntax of the assignment statement is

```
symbol = expression;
```

or

```
symbol op= expression;
```

where *op* is one of the operators +, -, *, or /. Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to *ld*(CP).

Link Editor Command Language

Assignment statements are normally placed outside the scope of section definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, (**.**), that can occur only within a section definition directive. This symbol refers to the current address of **ld(CP)**'s location counter. Thus, assignment expressions involving **.** are evaluated during the allocation phase of **ld(CP)**. Assigning a value to the **.** symbol within a section definition directive can increment (but not decrement) **ld(CP)**'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the **.** symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an n -byte boundary within an output section, where n is a power of 2. For example, the expression

`align(n)`

is equivalent to

`(. + n - 1) &~(n - 1)`

SIZEOF and **ADDR** are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

6

Link editor expressions may have either an absolute or a relocatable value. When **ld(CP)** creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.
- All other expressions are combinations of the above.

Specifying a Memory Configuration

MEMORY directives are used to specify

- the total size of the virtual space of the target machine
- the configured and unconfigured areas of the virtual space

If no directives are supplied, **ld**(CP) assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters \$, ., or -. Names of memory ranges are used by **ld**(CP) only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld**(CP)'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

- R : readable memory
- W : writable memory
- X : executable, i.e., instructions may reside in this memory
- I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

Link Editor Command Language

The syntax of the MEMORY directive is as follows:

```
MEMORY
{
    name1 (attr) :    origin = n1, length = n2
    name2 (attr) :    origin = n3, length = n4
    etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld**(CP) can be told that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

Section Definition Directives

6

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
    secname1 :
    {
        file_specifications,
        assignment_statements
    }
    secname2 :
    {
        file_specifications,
        assignment_statements
    }
    etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

```
filename ( secname )
```

or

```
filename ( secnam1 secnam2 . . . )
```

Sections of an input file are separated either by white space or commas, as are the file specifications themselves.

```
filename [COMMON]
```

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section.

Link Editor Command Language

For example,

```
SECTIONS
{
    outsec1:
    {
        file1.o (sec1)
        file2.o
        file3.o (sec1, sec2)
    }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

- section **sec1** from file **file1.o**
- all sections from **file2.o**, in the order they appear in the file
- section **sec1** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code

```
* (secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an **ld(CP)** option as shown in the following **SECTIONS** directive example:

```
SECTIONS
{
    outsec addr:
    {
        . . .
    }
    etc.
}
```


The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld(CP)** issues an appropriate error message. *addr* may also be the word **BIND**, followed by a parenthesized expression. The expression may use the pseudo-functions **SIZEOF**, **ADDR**, or **NEXT**. **NEXT** accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; **SIZEOF** and **ADDR** accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The **SECTIONS** directives defining output sections need not be given to **ld(CP)** in any particular order, unless **SIZEOF** or **ADDR** is used.

The **ld(CP)** does not ensure that the size of each section consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4.

The **ld(CP)** directives can be used to force a section to start on an odd byte boundary, although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld(CP)** issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The **ALIGN** option of the **SECTIONS** directive performs this function, so that the option

```
ALIGN (n)
```

is equivalent to specifying a bonding address of

```
( . + n - 1 ) & ~ ( n - 1 )
```

Link Editor Command Language

For example

```
SECTIONS
{
    outsec  ALIGN(0x20000) :
    {
        . . .
    }
    etc.
}
```

The output section **outsec** is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together

The default allocation algorithm for **ld(CP)** does the following:

- It links all input **.init** sections together, followed by **.text** sections, into one output section. This output section is called **.text** and is bound to an address of 0x0 plus the size of all headers in the output file.
- It links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine-dependent constant plus a number dependent on the size of headers and text.
- It links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**.

Specifying any **SECTIONS** directives results in this default allocation not being performed. Rather than relying on the **ld(CP)** default algorithm, if you are manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of **ld(CP)** is equivalent to supplying the following directive, where *align_value* is a machine-dependent constant.

For example,

```
SECTIONS
{
    .text sizeof_headers : { *(.init) *(.text) *(.fini))
    GROUP BIND( NEXT(align_value) +
    ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
    {
        .data      : { }
        .bss       : { }
    }
}
```

The GROUP command ensures that the two output sections, **.data** and **.bss**, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

For compatibility with UNIX System V Release 2, these addresses cannot change. Unfortunately, **.init** sections in the algorithm above will interfere with the placement of the signal recovery routines. Hence the **.text** sections are linked into the **a.out .text** section first. The **.init** sections (for shared libraries) and the **.fini** sections follow all of the **.text** sections. Routines in **crt1.0** branch to the **.init** sections before calling the **main()** function of the program.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following SECTIONS directive is used:

```
SECTIONS
{
    GROUP                                :
    {
        .text      : { }
        .data      : { }
        .bss       : { }
    }
}
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive.

Link Editor Command Language

To bind to 0xC0000, use

```
GROUP 0xC0000 : {
```

To align to 0x10000, use

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the **GROUP** directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
    .text      : { }
    .data ALIGN(0x400000) : { }
    .bss       : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x400000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld**(CP) cannot be used to force a certain allocation order in the output file.

6

Creating Holes Within Output Sections

The special symbol dot, (**.**), appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, **.** causes **ld**(CP)'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character, either the default fill character (0x00) or a supplied fill character. See the discussion of filling holes in "Initialized Section Holes or **.bss** Sections" in this chapter.

Consider the following section definition:

```
outsec:
{
    . += 0x1000;
    f1.o (.text)
    . += 0x100;
    f2.o (.text)
    . = align (F);
    f3.o (.text)
}
```

The effect of this command is as follows:

- A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.
- The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.
- The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld(CP)** treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(F) : {
```

Expressions that decrement **.** are illegal. For example, subtracting a value from the location counter is not allowed, since overwrites are not allowed. The most common operators in expressions that assign a value to **.** are **+=** and **align**.

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of **ld(CP)** can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. use of **.** to adjust **ld(CP)**'s location counter during allocation
2. use of **.** to assign an allocation-dependent value to a symbol
3. assigning an allocation-independent value to a symbol

Case 1 has already been discussed in the previous section.

Case 2 provides a means to assign addresses (known only after allocation) to symbols.

For example,

```
SECTIONS
{
    outsc1: {...}
    outsc2:
    {
        file1.o (s1)
        s2_start = . ;
        file2.o (s2)
        s2_end = . - 1;
    }
}
```

6

The symbol **s2_start** is defined to be the address of **file2.o(s2)**, and **s2_end** is the address of the last byte of **file2.o(s2)**.

Consider the following example:

```
SECTIONS
{
    outsc1:
    {
        file1.o (.data)
        mark = .;
        . += 4;
        file2.o (.data)
    }
}
```


In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving **.** must appear within **SECTIONS** definitions since they are evaluated during allocation. Assignment instructions that do not involve **.** can appear within **SECTIONS** definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld(CP)** issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a **MEMORY** directive). (The **>** notation is borrowed from the UNIX System concept of redirected output.)

For example,

```
MEMORY
{
    mem1:          o=0x000000      l=0x10000
    mem2 (RW):      o=0x020000      l=0x40000
    mem3 (RW):      o=0x070000      l=0x40000
    mem1:          o=0x120000      l=0x04000
}

SECTIONS
{
    outsec1: { f1.o(.data) } > mem1
    outsec2: { f2.o(.data) } > mem3
}
```

Link Editor Command Language

This directs **ld**(CP) to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld**(CP) normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a **SECTIONS** directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will apply:

- Explicit initialization options must be used to initialize all **.bss** sections in the output section.
- **ld**(CP) will use the default fill value to initialize all **.bss** sections in the output section.

Consider the following `ld(CP)` file:

```
SECTIONS
{
    sec1:
    {
        f1.o
        . =+ 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss) = 0x1234
    }
    sec3:
    {
        f3.o (.bss)
        . . .
    } = 0xFFFF
    sec4: { f4.o (.bss) }
}
```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within **sec3** as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section.

Notes and Special Considerations

Changing the Entry Point

The UNIX System **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

1. The value of the symbol specified with the **-e** option, if present, is used.
2. The value of the symbol **_start**, if present, is used.
3. The value of the symbol **main**, if present, is used.
4. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **-e** option or by using an assignment instruction in an ifile of the form

```
_start = expression;
```

6

If **ld(CP)** is called through **cc(CP)**, a startup routine is automatically linked in. Then, when the program is executed, the routine **exit(S)** is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld(CP)** directly or when changing the entry point. The user must supply the start-up routine or make sure that the program always calls **exit** rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar(CP)** command from object files generated by **cc** or **as**. **ar(CP)** is documented in the *Programmer's Reference*. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

- there exists a reference to a symbol defined in that member
- the reference is found by **ld**(CP) prior to the actual scanning of the library

When a library member is included by searching the library inside a **SECTIONS** directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside of a **SECTIONS** directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

- specific members of a library cannot be referenced explicitly in an ifile
- the default rules for the placement of members and sections cannot be overridden when they apply to archive library members

The **-l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **-l** option by simply giving the (full or relative) UNIX System file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld**(CP) will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

- The input files **file1.o** and **file2.o** each contain a reference to the external function **FCN**.
- Input **file1.o** contains a reference to symbol **ABC**.
- Input **file2.o** contains a reference to symbol **XYZ**.
- Library **liba.a**, member 0, contains a definition of **XYZ**.

Notes and Special Considerations

- Library **libc.a**, member 0, contains a definition of ABC.
- Both libraries have a member 1 that defines FCN.

If the **ld**(CP) command were entered as

```
ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by **liba.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified). If the **ld**(CP) command were entered as

```
ld file1.o file2.o -la -lc
```

then the FCN references are satisfied by **liba.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ is obtained from **liba.a**, member 0. If the **ld**(CP) command were entered as

```
ld file1.o file2.o -lc -la
```

then the FCN references are satisfied by **libc.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ is obtained from **liba.a**, member 0.

The **-u** option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

```
ld -u rout1 -la
```

creates an undefined symbol called **rout1** in **ld**(CP)'s global symbol table. If any member of library **liba.a** defines this symbol, it is extracted (and perhaps other members are extracted as well). Without the **-u** option, there would have been no unresolved references or undefined symbols to cause **ld**(CP) to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined in such a way that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
    mem1:          o = 0x00000          1 = 0x02000
    mem2:          o = 0x40000          1 = 0x05000
    mem3:          o = 0x20000          1 = 0x10000
}
```

Let the files **f1.o**, **f2.o**, . . . **fn.o** each contain three sections **.text**, **.data**, and **.bss**, and suppose the combined **.text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so **ld(CP)** may do allocation. For example,

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a **SECTIONS** directive, by combining input sections of the same name, or by combining **.text** and **.init** into **.text**. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. **ld(CP)** uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.
3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no **SECTIONS** directives are given, then output sections are allocated in the order they appear to **ld(CP)**. Otherwise, output sections are allocated in the order they were defined or made known to **ld(CP)** and put into the first available space they will fit in.

Incremental Link Editing

As previously mentioned, the output of **ld(CP)** can be used as an input file to subsequent **ld(CP)** runs providing that the relocation information is retained (**-r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. For example,

Step 1:

```
ld -r -o outfile1 ifile1 infile1.o
```

```
/* ifile1 */
SECTIONS
{
    ssl:
    {
        f1.o
        f2.o
        . . .
        fn.o
    }
}
```

Step 2:

```
ld -r -o outfile2 ifile2 infile2.o
```

```
/* ifile2 */
SECTIONS
{
    ss2:
    {
        g1.o
        g2.o
        . . .
        gn.o
    }
}
```

Step 3:

```
ld -a -o final.out outfile1 outfile2
```

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules:

- Intermediate link edits should contain only **SECTIONS** declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
- All allocation and memory directives, as well as any assignment statements, are included only in the final **ld(CP)** call.

DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
    name1 0x200000 (DSECT)      : { file1.o }
    name2 0x400000 (COPY)       : { file2.o }
    name3 0x600000 (NOLOAD)     : { file3.o }
    name4          (INFO)       : { file4.o }
    name5 0x900000 (OVERLAY)    : { file5.o }
}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

- It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by `ld(CP)`.
- It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
- The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).
- None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from `file1.o` are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file, whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct ld(CP) to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
    .text BLOCK(0x200) : { }
    .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, ld(CP) assures that each section, .text and .data, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

Nonrelocatable Input Files

If a file produced by **ld**(CP) is intended to be used in a subsequent **ld**(CP) run, the first **ld**(CP) run should have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld**(CP) does not have relocation or symbol table information [perhaps from the action of a **strip**(CP) command, or from being link edited without a **-r** option, or with a **-s** option], the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

- Each input file must have no unresolved external references.
- Each input file must be bound to the exact same virtual address as it was bound to in the **ld**(CP) run that created it.

Note

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld**(CP).

Syntax Diagram for Input Directives

Figure 6-2 Syntax Diagram for Input Directives (Sheet 1 of 4)

Directives	Expanded Directives
<infile>	{<cmd>}
<cmd>	<memory> <sections> <assignment> <filename> <flags>
<memory>	MEMORY { <memory_spec> { [,] <memory_spec> } }
<memory_spec>	<name> [<attributes>] : <origin_spec> [,] <length_spec>
<attributes>	({ R W X I })
<origin_spec>	<origin> = <long>
<length_spec>	<length> = <long>
<origin>	ORIGIN o org origin
<length>	LENGTH l len length

6

Note

Two punctuation symbols, brackets and braces, do double duty in this diagram.

Where the actual symbols, [] and { } are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols [and] (larger and in bold), it means the material enclosed is optional.

Where you see the symbols { and } (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Figure 6-2 Syntax Diagram for Input Directives (Sheet 2 of 4)

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> <group> <library>
<group>	GROUP <group_options> : { <section_list> } [<mem_spec>]
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> : { <statement> } [<fill>] [<mem_spec>]
<group_options>	[<addr>] [<align_option>] [<block_option>]
<sec_options>	[<addr>] [<align_option>] [<block_option>] [<type_option>]
<addr>	<long> <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN align
<block_option>	<block> (<long>)
<block>	BLOCK block
<type_option>	(DSECT) (NOLOAD) (COPY) (INFO) (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name> > <attributes>
<statement>	<filename> <filename> (<name_list>) [COMMON] * (<name_list>) [COMMON] <assignment> <library> <i>null</i>

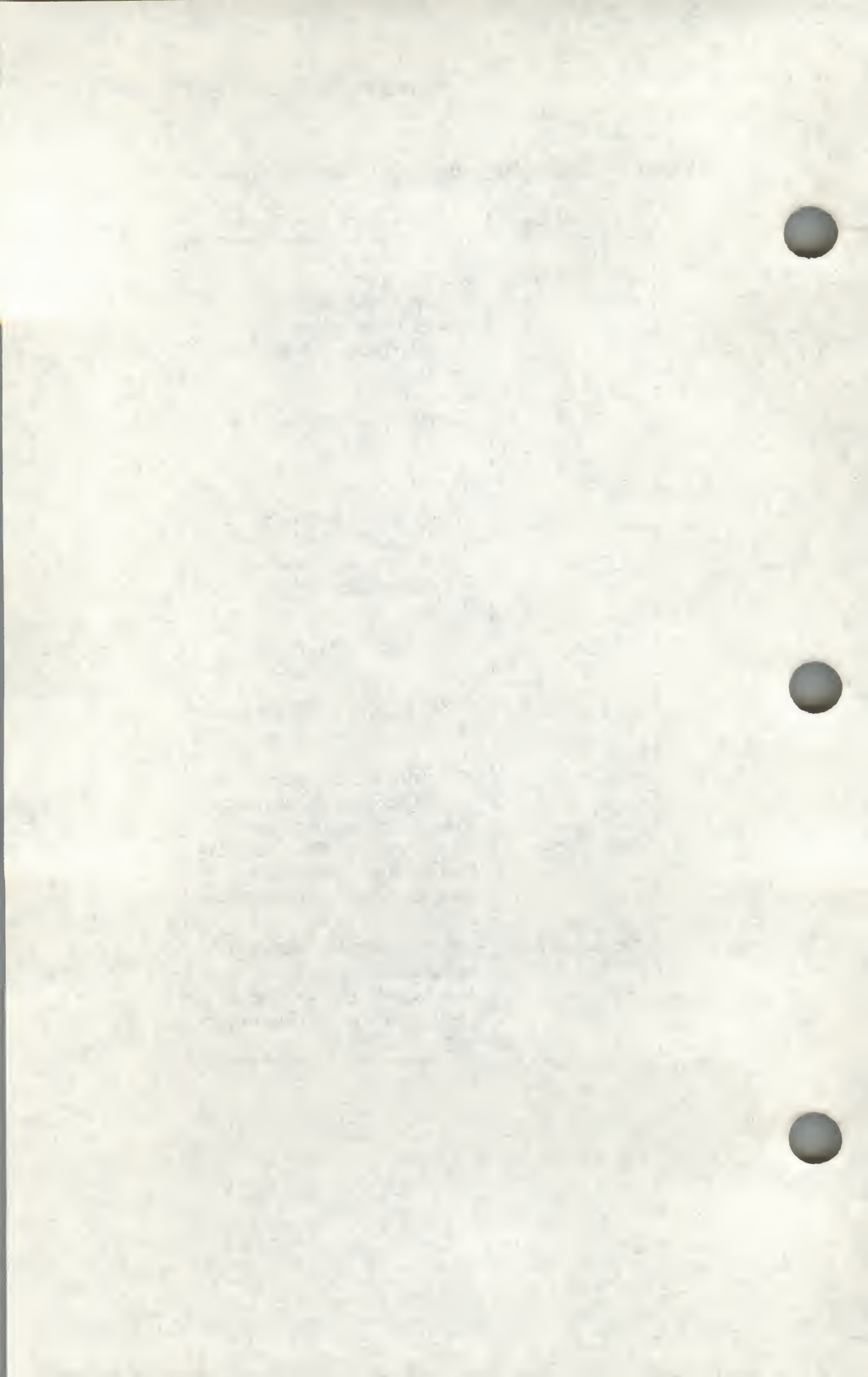
Figure 6-2 Syntax Diagram for Input Directives (Sheet 3 of 4)

Directives	Expanded Directives
<name_list>	<section_name> [,] { <section_name> }
<library>	-!<name>
<bind>	BIND bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> .
<assign_op>	= += -= *= /=
<end>	; ,
<expr>	<expr> <binary_op> <expr>
<binary_op>	* / % + - >> << == != > < <= >= & &&
<term>	<long> <name> <align> (<term>) (<expr>) <unary_op> <term> <phy> (<lside>) <sizeof>(<sectionname>) <next>(<long>) <addr>(<sectionname>)
<unary_op>	! -
<phy>	PHY phy
<sizeof>	SIZEOF sizeof

Syntax Diagram for Input Directives

Figure 6-2 Syntax Diagram for Input Directives (Sheet 4 of 4)

Directives	Expanded Directives
<next>	NEXT next
<addr>	ADDR addr
<flags>	-e<whitespace><name> -f<whitespace><long> -h<whitespace><long> -l<name> -m -o<whitespace><filename> -r -s -t -u<whitespace><name> -z -H -L<path_name> -M -N -S -V -VS<whitespace><long> -a -x
<name>	Any valid symbol name
<long>	Any valid long integer constant
<whitespace>	Blanks, tabs, and newlines
<filename>	Any valid UNIX Operating System file name. This may include a full or partial path name.
<sectionname>	Any valid section name, up to 8 characters
<path_name>	Any valid UNIX Operating System path name (full or partial)



Chapter 7

Source Code Control System (SCCS)

Introduction 7-1

SCCS for Beginners 7-2

Terminology 7-2

Creating an SCCS File by Means of **admin** 7-2

Retrieving a File by Means of **get** 7-3

Recording Changes by Means of **delta** 7-4

Additional Information about **get** 7-5

The **help** Command 7-6

Delta Numbering 7-7

SCCS Command Conventions 7-10

x.files and *z.files* 7-11

Error Messages 7-11

SCCS Commands 7-12

The **get** Command 7-13

The **delta** Command 7-24

The **admin** Command 7-27

Creation of SCCS Files 7-27

The **prs** Command 7-30

The **sact** Command 7-32

The **rmDEL** Command 7-32

The **cdc** Command 7-33

The **what** Command 7-34

The **sccsdiff** Command 7-34

The **comb** Command 7-35

The **val** Command 7-36

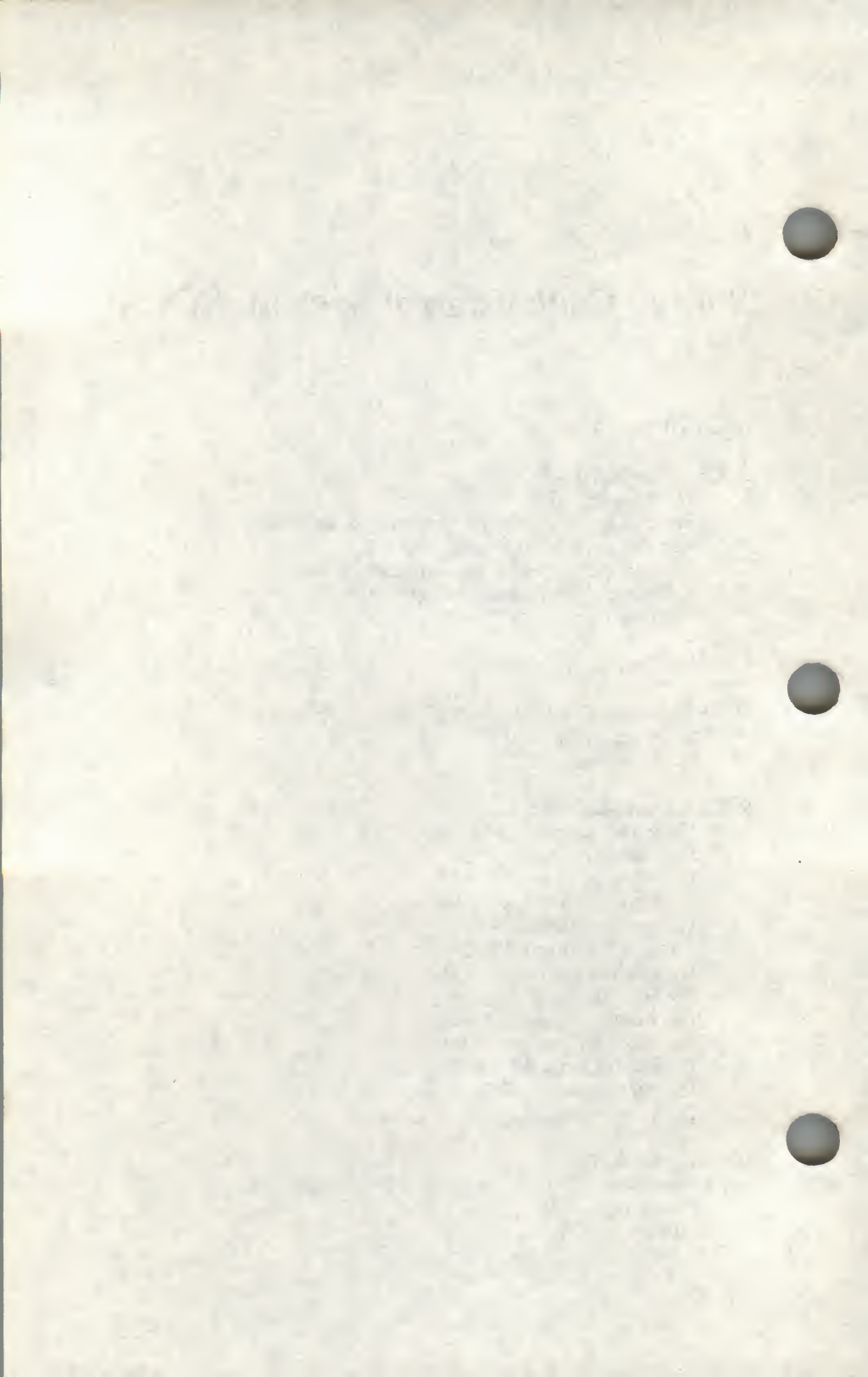
The **vc** Command 7-36

SCCS Files 7-37

Protection 7-37

Formatting 7-38

Auditing 7-39



Introduction

The Source Code Control System (SCCS) is a maintenance and enhancement tracking tool that runs under the UNIX System. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version to remember why the changes were made, who made them, and when they were made.

This guide covers the following:

- **SCCS for Beginners:** how to make, retrieve, and update an SCCS file
- **Delta Numbering:** how versions of an SCCS file are named
- **SCCS Command Conventions:** what rules apply to SCCS commands
- **SCCS Commands:** the fourteen SCCS commands and their more useful arguments
- **SCCS Files:** protection, format, and auditing of SCCS files

Neither the implementation of SCCS nor its installation procedure is described in this guide.

SCCS for Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS Identification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file (that is, its first delta) is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. There will be more on delta numbering later. At this point, it is enough to know that, by default, SCCS assigns SIDs automatically.

Creating an SCCS File by Means of `admin`

Suppose, for example, you have a file called **lang** that is simply a list of five programming language names. Use a text editor to create file **lang** containing the following list:

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** command (that is, administer SCCS file). The following creates an SCCS file from the **lang** file:

```
admin -ilang s.lang
```

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **-i** key letter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies:

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get** command under “SCCS Commands.” In the following examples, this warning message is not shown, although it may be issued.

Remove the **lang** file. It is no longer needed because it exists now under SCCS as **s.lang**.

```
rm lang
```

Retrieving a File by Means of **get**

Use the **get** command as follows:

```
get s.lang
```

This retrieves **s.lang** and prints:

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text has been placed in a new file known as a “g.file.” SCCS forms the *g.file* name by deleting the prefix **s.** from the name of the SCCS file. Thus, the original **lang** file has been recreated.

If you list **ls(C)** (documented in the *User's Reference*) the contents of your directory, you will see both **lang** and **s.lang**. SCCS retains **s.lang** for other users.

The **get s.lang** command creates **lang** as read-only and keeps no information regarding its creation. Because you are going to make changes to it, **get** must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The **get -e** command causes SCCS to create **lang** for both reading and writing (editing). It also places certain information about **lang** in another new file, called the “p.file” (**p.lang**, in this case), which is needed later by the **delta** command.

SCCS for Beginners

The **get -e** command prints the same messages as **get**, except that now the SID for the first delta you will create is issued:

```
1.1
new delta 1.2
5 lines
```

Change **lang** by adding two more programming languages:

```
SNOBOL
ADA
```

Recording Changes by Means of delta

Next, use the **delta** command as follows:

```
delta s.lang
```

Then **delta** prompts with:

```
comments?
```

Your response should be an explanation of why the changes were made. For example:

```
added more languages
```

The **delta** command now reads the p.file **p.lang** and determines what changes you made to **lang**. It does this by doing its own **get** to retrieve the original version and applying the **diff(C)** command, documented in the *User's Reference*, to the original version and the edited version. Next, **delta** stores the changes in **s.lang** and destroys the no longer needed **p.lang** and **lang** files.

When this process is complete, **delta** outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the **delta** you just created, and the next three lines summarize what was done to **s.lang**.

Additional Information about get

The command:

```
get s.lang
```

retrieves the latest version of the file **s.lang**, 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, any of the following will retrieve version 1.2:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following **-r** are SIDs. When you omit the level number of the SID (as in **get -r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version of release 1, 1.2. The third command specifically requests the retrieval of a particular version, in this case also, 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version before release 2. The **get** command also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is:

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version **delta** will create.

SCCS for Beginners

If, for example, the file is now edited by deleting COBOL from the list of languages, and **delta** is executed as follows:

```
delta s.lang  
comments? deleted cobol from list of languages
```

then you will see by **delta**'s output that version 2.1 is indeed created:

```
2.1  
0 inserted  
1 deleted  
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, and so on.), or another new release can be created in a similar manner.

The help Command

If the command:

```
get lang
```

is now executed, it will generate the following message, as follows:

```
ERROR [lang]: not an SCCS file (co1)
```

The code **co1** can be used with **help** to print a fuller explanation of the message:

```
help co1
```

This gives the following explanation of why **get lang** produced an error message:

```
co1:  
"not an SCCS file"  
A file that you think is an SCCS file  
does not begin with the characters "s".
```

The **help** command is useful almost any time there is doubt about the meaning of an SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, and so on. The components of these SIDs are called “release” and “level numbers”, respectively. Thus, normal naming of new deltas proceeds by incrementing the level numbers. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the number then applies to all new deltas, unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 7-1.

Figure 7-1 Evolution of an SCCS File



This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned for a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 7-1. Assume that a production user reports a problem in version 1.3 that cannot wait for correction until release 2. The changes necessary to correct the problem will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will not affect the changes being applied for release 2 (that is, deltas 1.4, 2.1, 2.2, and so on). This new delta is the first node of a new branch of the tree.

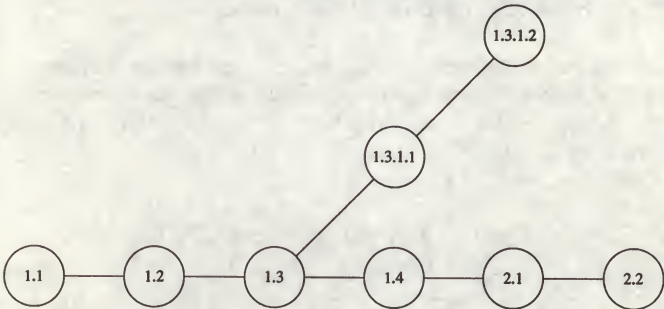
Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

Delta Numbering

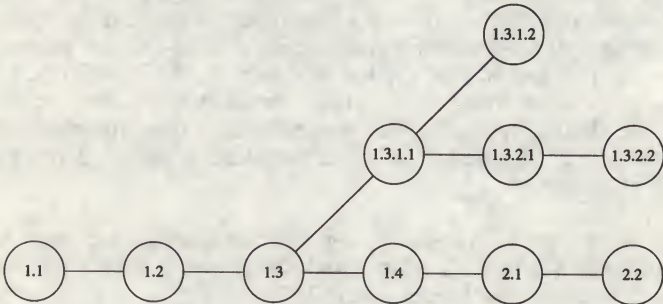
The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, and so forth. This is shown in Figure 7-2:

Figure 7-2 Tree Structure with Branch Deltas



The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 7-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, and so on), but the secondary branch number remains the same.

Figure 7-3 Extended Branching Concept



The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments:

- key letters
- file names

A key letter is an option that begins with a minus sign, `-`, followed by a lowercase letter and, in some cases, a value.

File and directory names specify the file(s) that the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes by means of `chmod(C)`, documented in the *User's Reference*] in the directories named are silently ignored.

In general, a filename argument may not begin with a minus sign. If a filename `-` (a lone minus sign) is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands `find(C)` and `ls(C)`, which are also documented in the *User's Reference*.

Key letters are processed before file names. Therefore, the placement of key letters is arbitrary; that is, they may be interspersed with file names. File names, however, are processed left to right. Somewhat different conventions apply to `what(CP)`, `sccsdiff(CP)`, and `val(CP)`, detailed later under "SCCS Commands" and documented in the *Programmer's Reference*.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but, for a complete description, see `admin(CP)` in the *Programmer's Reference*.

The distinction between real user (see `passwd(C)`) and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are both the person logged into the UNIX System.

x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the “x.file.” This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x.file by replacing the s. of the SCCS file name with x.. The x.file is created in the same directory as the SCCS file, given the same mode (see **chmod(C)** in the *User’s Reference*), and owned by the effective user. When processing is complete, the old SCCS file is destroyed, and the modified x.file is renamed (with x. relaced by s.) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the “z.file.” SCCS forms its name by replacing the s. of the SCCS file name with a z. prefix. The z.file contains the process number of the command that created it, and its existence prevents other commands from processing the SCCS file. The z.file is created with access permission mode 444 (read only) in the same directory as the SCCS file, and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, users can ignore x.files and z.files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments. Full descriptions with details of all arguments are in the *Programmer's Reference*.

Here is a quick-reference overview of the commands:

get	retrieves versions of SCCS files.
unget	undoes the effect of a get -e before the file is deltaed .
delta	applies deltas (changes) to SCCS files and creates new versions.
admin	initializes SCCS files, manipulates their descriptive text, and controls delta creation rights.
prs	prints portions of an SCCS file in user-specified format.
sact	prints information about files that are currently out for editing.
rmDEL	removes a delta from an SCCS file; allows removal of deltas created by mistake.
cdc	changes the commentary associated with a delta.
what	searches any UNIX System file(s) for all occurrences of a special pattern and prints out what follows it; useful in finding identifying information inserted by the get command.
scsdiff	shows differences between any two versions of an SCCS file.
comb	combines consecutive deltas into one to reduce the size of an SCCS file.
val	validates an SCCS file.
vc	a filter that may be used for version control.

The get Command

The **get**(CP) command creates a file containing a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the "g.file." It is created in the current directory and owned by the real user. The mode assigned to the g.file depends on how the **get** command is used.

The most common use of **get** is:

```
get s.abc
```

which normally retrieves the latest version of file *abc* from the SCCS file tree trunk and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

This means version 1.3 of file *s.abc* has been retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords have been substituted in the file.

The generated g.file (file *abc*) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce g.files only for inspection, compilation, and so on. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example,

```
get s.abc s.xyz
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```


SCCS Commands

ID Keywords

In generating a g.file for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name within the g.file. This information eventually appears in a load module when one is created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) key words appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example,

%I%

is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, **%H%** and **%M%** are the names of the g.file. Thus, executing **get** on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/18/85');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately twenty ID keywords provided, see **get(CP)** in the *Programmer's Reference*.

Retrieval of Different Versions

The version of an SCCS file that **get** retrieves is the most recently created delta of the highest-numbered trunk release. However, any other version can be retrieved with **get -r** by specifying the version's SID.

Thus:

```
get -r1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file *s.abc* and release 7 is the highest-numbered release below 9. Executing:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9.

SCCS Commands

Similarly, omitting the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8  
89 lines
```

The **get -t** command will retrieve the latest (top) version of a particular release when no **-r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5:

```
get -r3 -t s.abc
```

might produce:

```
3.5  
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5  
46 lines
```

7

Retrieval With Intent to Make a Delta

The **get -e** command indicates an intent to make a delta. First, **get** checks the following conditions.

1. It checks whether the login name or group ID of the person executing **get** is present in the user list. The login name or group ID must be present for the user to be allowed to make deltas. (See “The **admin** Command” for a discussion of making user lists.)

2. It checks whether the release number (R) of the version being retrieved satisfies the relation:

```
floor is less than or equal to R,
which is less than or equal to ceiling.
```

This check determines whether the release being accessed is protected. The floor and ceiling are flags in the SCCS file, representing start and end of range.

3. It checks whether the R is locked against editing. The lock is a flag in the SCCS file.
4. It checks whether multiple concurrent edits are allowed for the SCCS file by the j flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get -e** causes the creation of a g.file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g.file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g.file while it is being edited for the purpose of making a delta.

Any ID keywords appearing in the g.file are not substituted by **get -e** because the generated g.file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the g.file. Thus, the message:

```
No id keywords (cm7)
```

is never output when **get -e** is used.

In addition, **get -e** causes the creation (or updating) of a p.file that is used to pass information to the **delta** command.

The following:

```
get -e s.abc
```

produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

SCCS Commands

Undoing a `get -e`

There may be times when a file is erroneously retrieved for editing, when there is really no editing that needs to be done at the time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional `get` Options

If `get -r` and/or `-t` are used together with `-e`, the version retrieved for editing is the one specified with `-r` and/or `-t`.

The `get -i` and `-x` commands are used to specify a list of deltas to be included and excluded, respectively. (See `get(CP)` in the *Programmer's Reference* for the syntax of such a list.) Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, `get` checks for possible interference with other deltas. For example, two deltas can interfere when each one changes the same line of the retrieved `g.file`. A warning shows the range of lines within the retrieved `g.file` where the problem may exist. The user should examine the `g.file` to determine what the problem is and take appropriate corrective steps (such as edit the file). The `get -i` and `get -x` commands should be used with extreme care.

The `get -k` command is used either to regenerate a `g.file` that may have been accidentally removed or ruined after `get -e`, or simply to generate a `g.file` in which the replacement of ID keywords has been suppressed. A `g.file` generated by `get -k` is identical to one produced by `get -e`, but no processing related to the `p.file` takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several `get -e` commands may be executed on the same file unless two executions retrieve the same version or multiple concurrent edits are allowed.

The p.file created by `get -e` is named by automatic replacement of the SCCS filename's prefix `s.` with `p.`. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p.file contains the following information for each delta that is still in progress:

- the SID of the retrieved version
- the SID given to the new delta when it is created
- the login name of the real user executing `get`

The first execution of `get -e` causes the creation of a p.file for the corresponding SCCS file. Subsequent executions simply update the p.file with a line containing the above information. Before updating, however, `get` checks to assure that the SID of the version to be retrieved has not already been retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress, and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of `get` must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the g.file, which is an SCCS error condition. In practice, this problem does not arise because each user normally has a different working directory. See "Protection" under "SCCS Files" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 7-4 shows the possible SID components a user can specify with `get` (left-most column), the version that will then be retrieved by `get`, and the resulting SID for the delta, which `delta` will create (right-most column).

Figure 7-4 Determination of New SID

SID Specified in get*	-b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta to be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	—	R < mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	—	Trunk successor number in release > R, and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB+1).1
Footnotes *, †, ‡, §, and ** on next page.				

Footnotes to Figure 7-4:

- * R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (that is, the maximum branch number plus 1) of level L within release R. Note that, if the SID specified is R.L, R.L.B, or R.L.B.S, then each of these specified SID numbers must exist.
- † The **-b** key letter is effective only if the **b** flag (discussed under **admin**(CP) in the *Programmer's Reference*) is present in the file. An entry of **-** means "irrelevant".
- ‡ This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as though specified on the command line. Thus, one of the other cases in this figure applies.
- § This is used to force the creation of the first delta in a new release.
- ** The release hR is the highest existing release that is lower than the specified, nonexistent release R.

Concurrent Edits of Same SID

Under normal conditions, more than one **get -e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get -e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

SCCS Commands

without an intervening **delta**. In this case, a **delta** after the first **get** will produce delta 1.2 (assuming that 1.1 is the most recent trunk delta), and a **delta** after the second **get** will produce delta 1.1.1.1.

Key Letters That Affect Output

The **get -p** command causes the retrieved text to be written to the standard output, rather than to a *g*.file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. The **get -p** command is used, for example, to create a *g*.file with an arbitrary name, as in:

```
get -p s.abc > arbitrary-file-name
```

The **get -s** command suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. The **get -s** command is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used with **-p** to pipe the output, as in:

```
get -p -s s.abc | pg
```

The **get -g** command suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file,

```
get -g -r4.3 s.abc
```

7 outputs the SID 4.3 if it exists in the SCCS file *s.abc*, or an error message if it does not. Another use of **get -g** is in regenerating a *p*.file that may have been accidentally destroyed, as in:

```
get -e -g s.abc
```

The **get -l** command causes SCCS to create an "l.file." It is named by replacing the *s*. of the SCCS filename with *l*., created in the current directory with mode 444 (read only) and owned by the real user. The *l*.file contains a table (whose format is described under **get(CP)** in the *Programmer's Reference*) showing the deltas used in constructing a particular version of the SCCS file.

For example:

```
get -r2.3 -l s.abc
```

generates an l.file showing the deltas applied to retrieve version 2.3 of file s.abc.

Specifying p with -l, as in:

```
get -lp -r2.3 s.abc
```

causes the output to be written to the standard output, rather than to the l.file. The **get -g** command can be used with **-l** to suppress the retrieval of the text.

The **get -m** command identifies the changes applied to an SCCS file. Each line of the g.file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

The **get -n** command causes each line of a g.file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep(C)**, which is documented in the *User's Reference*. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both **-m** and **-n** are specified, each line of the generated g.file is preceded by the value of the **chap3.13** ID keyword and a tab (the effect of **-n**), and is followed by the line in the format produced by **-m**. Because use of **-m** and/or **-n** causes the contents of the g.file to be modified, such a g.file must not be used for creating a delta. Therefore, neither **-m** nor **-n** may be specified together with **get -e**.

Note

See **get(CP)** in the *Programmer's Reference* for a full description of additional key letters.

The delta Command

The **delta**(CP) command is used to incorporate changes made to a g.file into the corresponding SCCS file (that is, to create a delta and, therefore, a new version of the file).

The **delta** command requires the existence of a p.file (created by means of **get -e**). It examines the p.file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

The **delta** command performs the same permission checks that **get -e** performs. If all checks are successful, **delta** determines what has been changed in the g.file by comparing it using **diff(C)**, documented in the *User's Reference*, with its own temporary copy of the g.file as it was before editing. This temporary copy of the g.file is called the d.file, and is obtained by performing an internal **get** on the SID specified in the p.file entry.

The required p.file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g.file must be the one who creates the delta. However, if the login name of the user appears in more than one entry, then the same user has executed **get -e** more than once on the same SCCS file. Then, **delta -r** must be used to specify the SID that uniquely identifies the p.file entry. This entry becomes the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is:

```
delta s.abc
```

which prompts:

```
comments?
```

to which the user replies with a description of why the delta is being made, ending the reply with a new-line character. The user's response may be up to 512 characters long with new-lines (not intended to terminate the response) escaped by backslashes (\).

If the SCCS file has a **v** flag, **delta** first prompts with:

```
MRs?
```

which stands for Modification Requests, on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a new-line character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled

environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, or the like, collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change-management process.

The **delta -y** and/or **delta -m** commands can be used to enter comments and MR numbers on the command line rather than through the standard input, as in:

```
delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two key letters are useful when **delta** is executed from within a shell procedure. (See **sh(C)** in the *User's Reference*.)

Note

The **delta -m** command is allowed only if the SCCS file has a **v** flag.

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p.file) and the number of lines inserted, deleted, and left unchanged. Here is an example:

```
1.4
14 inserted
7 deleted
345 unchanged
```


SCCS Commands

If line counts do not agree with the user's perception of the changes made to a g.file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g.file. However, the total number of lines new delta's (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g.file.

If you are in the process of making a delta and the **delta** command finds no ID keywords in the edited g.file, the message:

```
No id keywords (cm7)
```

is issued for commentary after the prompts but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a g.file that was created by a **get** without **-e**. (ID keywords are replaced by **get** in such a case.) It could also be caused by accidentally deleting or changing ID keywords while editing the g.file. A third possibility is that the file had no ID keywords. In any case, the delta will be created unless there is an **i** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding p.file entry is removed from the p.file. All updates to the p.file are made to a temporary copy, the "q.file," whose use is similar to that of the x.file described earlier under "SCCS Command Conventions." If there is only one entry in the p.file, then the p.file itself is removed.

In addition, **delta** removes the edited g.file unless **-n** is specified. For example:

```
delta -n s.abc
```

will keep the g.file after processing.

The **delta -s** command suppresses all output normally directed to the standard output, other than **comments?** and **MRs?**. Thus, use of **-s** with **-y** (and/or **-m**) causes **delta** not to read the standard input or write the standard output.

The changes made to the g.file constitute the delta and may be printed on the standard output by using **delta -p**. The format of this output is similar to that produced by **diff(C)**, documented in the *User's Reference*.

The admin Command

The **admin**(CP) command, documented in the *Programmer's Reference*, is used to administer SCCS files; that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of key letters with **admin**, or are assigned default values if no key letters are supplied. The same key letters are used to change the parameters of existing SCCS files.

Two key letters are used in detecting and correcting corrupted SCCS files. (See "Auditing" under "SCCS Files".)

Newly-created SCCS files are given access permission mode 444 (read only), and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command:

```
admin -ifirst s.abc
```

in which the value **first** with **-i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **-i** means **admin** is to read the standard input for the text of the initial delta.

The command:

```
admin -i s.abc < first
```

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the command also sets the **i** flag (not to be confused with the **-i** key letter), then the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin -i**.

SCCS Commands

admin -r is used to specify a release number for the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

means that the first delta should be named 3.1 rather than the normal 1.1. Because **-r** has meaning only when creating the first delta, its use is permitted only with **-i**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin -y**) and/or MR numbers (**-m**) can be entered in exactly the same way as a **delta**.

If **-y** is omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**admin -m**), the **v** flag must be set by means of **-f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary. (See **scsfile(F)** in the *Programmer's Reference*] in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that **-y** and **-m** are effective only if a new SCCS file is being created.

7

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin -t**.

When an SCCS file is first being created and **-t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file **desc**.

When processing an existing SCCS file, **-t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

admin -tdesc s.abc

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*. Omission of the file name after the **-t** key letter, as in:

admin -t s.abc

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **admin -f** or deleted by means of **-d**.

SCCS file flags are used to direct certain actions of the various commands. (See **admin(CP)** in the *Programmer's Reference* for a description of all the flags.) For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

The **admin -f** command is used to set flags and, if desired, their values. For example:

admin -ifirst -fi -fmmodname s.abc

sets the **i** and **m** (module name) flags. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the **%M%** ID keyword.) Several **-f** key letters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

The **admin -d** command is used to delete a flag from an existing SCCS file. As an example, the command:

admin -dm s.abc

removes the **m** flag from the SCCS file. Several **-d** key letters may be used with one **admin** and may be intermixed with **-f**.

SCCS Commands

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), use **admin -a**. For example,

```
admin -axyz -awql -a1234 s.abc
```

adds the login names **xyz** and **wql** and the group ID **1234** to the list. The **admin -a** command may be used whether creating a new SCCS file or processing an existing one.

The **admin -e** command (erase) is used to remove login names or group IDs from the list.

The prs Command

The **prs**(CP) command is used to print all or part of an SCCS file on the standard output. If **prs -d** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data key words (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values, according to their definitions. For example:

```
:I:
```

is defined as the data keyword replaced by the SID of a specified delta. Similarly, **:F:** is the data keyword for the SCCS file name currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data key word. For a complete list, see **prs**(CP) in the *Programmer's Reference*.

There is no limit to the number of times a data key word may appear in a data specification. Thus, for example,

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **prs -r**. For example,

```
prs -d":F:::I: comment line is :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **-r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **-l** or **-e**. Using **prs -e** substitutes data keywords for the SID designated by means of **-r** and all deltas created earlier, while **prs -l** substitutes data keywords for the SID designated by means of **-r** and all deltas created later. Thus, the command:

```
prs -d:I: -r1.4 -e s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command:

```
prs -d:I: -r1.4 -l s.abc
```

may produce:

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **-e** and **-l**.

The **sact** Command

The **sact**(CP) command is like a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get -e** command, and the date and time the **get -e** was executed. It is a useful command for an administrator. It is described in the *Programmer's Reference*.

The **rmidel** Command

The **rmidel**(CP) command, documented in the *Programmer's Reference*, allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 7-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **-r** key letter is mandatory with **rmidel**. It is used to specify the complete SID of the delta to be removed. Thus:

```
rmidel -r2.3 s.abc
```

specifies the removal of trunk delta 2.3.

Before removing the delta, **rmidel** checks that the release number (R) of the given SID satisfies the relation:

floor less than or equal to R less than or equal to ceiling

The **rmidel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated delta has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the **l** flag is set the release must not be contained in the list. (See **admin**(CP) in the *Programmer's Reference*.) If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

The `cdc` Command

The `cdc`(CP) command, documented in the *Programmer's Reference*, is used to change the commentary made when the delta was created. It is similar to the `rmDEL` command (for instance, `-r` and full SID are necessary), although the delta need not be a leaf delta. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for, as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing `cdc` and the time of its execution.

The `cdc` command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus,

```
cdc -r1.4 s.abc
```

```
MRs? mrnum3 !mrnum1
```

(The MRs? prompt appears only if the `v` flag has been set.)

```
comments?
```

deleted wrong MR number and inserted correct MR number

inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

Note

An MR (Modification Request) is described in "The **delta** Command" section of this guide.

The what Command

The **what**(CP) command, described in the *Programmer's Reference*, is used to find identifying information within any UNIX System file whose name is given as an argument. No key letters are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword. (See **get**(CP) in the *Programmer's Reference*.) It prints on the standard output whatever follows the string until the first double quote (**"**), greater than (**>**), backslash (****), new-line, or nonprinting NUL character.

For example, if an SCCS file called **s.prog.c** (a C language program) contains the following line:

```
char id[] = "%W%";
```

and the command:

```
get -r3.4 s.prog.c
```

is used, the resulting g.file is compiled to produce **prog.o** and **a.out**. Then, the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
  prog.c: 3.4
prog.o:
  prog.c: 3.4
a.out:
  prog.c: 3.4
```

The string searched for by **what** need not be inserted by means of an ID keyword of **get**; you may insert it in any convenient manner.

The sccsdiff Command

The **sccsdiff**(CP) command, documented in the *Programmer's Reference*, determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff -r** in the same way as with **get -r**. SID numbers must be specified as the first two arguments. Any following key letters are interpreted as arguments to the **pr**(C) command, documented in the *User's Reference* (which prints the differences), and must appear before any file names. The SCCS file(s) to be processed are named last.

Directory names and a name “-” (a lone minus sign) are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

The differences are printed the same way as by **diff(C)**, which is discussed in the *User's Reference*.

The comb Command

The **comb(CP)** command, documented in the *Programmer's Reference*, lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh(C)** in the *User's Reference*) on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any key letters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 7-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the key letters used with this command are as follows:

- comb -s** This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.
- comb -p** This option is used to specify the oldest delta the user wants preserved.
- comb -c** This option is used to specify a list (see **get(CP)** in the *Programmer's Reference* for its syntax) of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val Command

The **val**(CP) command (see the *Programmer's Reference*) is used to determine whether a file is an SCCS file meeting the characteristics specified by certain key letters. It checks for the existence of a particular delta when the SID for that delta is specified with **-r**.

The string following **-y** or **-m** is used to check the value set by the **t** or the **m** flag, respectively. See **admin**(CP) in the *Programmer's Reference* for descriptions of these flags.

The **val** command treats the special argument **-** differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file ((CTL)D) is entered. This permits one **val** command with different values for key letters and file arguments. For example:

```
val - -yc -mabc s.abc -mxyz -ypl1 s.xyz
```

first checks whether file **s.abc** has a value **c** for its type flag and value **abc** for the module name flag. Once this is done, **val** processes the remaining file, in this case, **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error. (See **val**(CP) for a description of errors and codes.) In addition, an appropriate diagnostic is printed, unless suppressed by **-s**. A return code of 0 means all files met the characteristics specified.

The vc Command

7

The **vc**(CP) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *Programmer's Reference*.

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the UNIX System for most of the protection mechanisms required to prevent unauthorized changes to SCCS files; that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the x.file; see “SCCS Command Conventions”). When processing is done, the old file is automatically removed and the x.file, renamed (with an s. prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands will produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (such as on large development projects), one user (that is, one user ID) must be chosen as the owner of the SCCS files. This person will administer the files (use the **admin** command and so forth) and be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent

SCCS Files

program is required to provide an interface to the **get**, **delta**, and, if desired, **rmDEL** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the set-user-ID-on-execution bit on. (See **chmod(C)** in the *User's Reference*.) This assures that the effective user ID is that of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmDEL** and **cdc**.

A project-dependent interface program, as its name implies, can be custom-built for each project.

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	This is a line containing the logical sum of all the characters of the file (not including the checksum itself).
Delta Table	This provides information about each delta, such as type, SID, date and time of creation, and commentary.
User Names	This is the list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	These are indicators that control certain actions of SCCS commands.
Descriptive Text	This is usually a summary of the contents and purpose of the file.
Body	This is the text administered by SCCS, intermixed with internal SCCS control lines.

Details on these file sections may be found in **sccsfile(F)**. The checksum is discussed below under "Auditing."

Since SCCS files are ASCII files, they can be processed by non-SCCS commands like **ed(C)**, **grep(C)**, and **cat(C)**, all documented in the *User's Reference*. This is convenient when an SCCS file must be modified manually (such as when a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.

Note

Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with **ed(C)**]. No SCCS command will process a corrupted SCCS file except the **admin** command with **-h** or **-z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin -h** and specify all SCCS files:

```
admin -h s.file1 s.file2 ...
      or
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
corrupted file (co6)
```

SCCS Files

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the `ls(C)` command, described in the *User's Reference*, periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX System operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

admin -z s.file

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

Chapter 8

m4: A Macro Processor

Introduction 8-1

Invoking m4 8-2

Defining Macros 8-3

Quoting 8-5

Using Arguments 8-7

Using Built-in Arithmetic Values 8-8

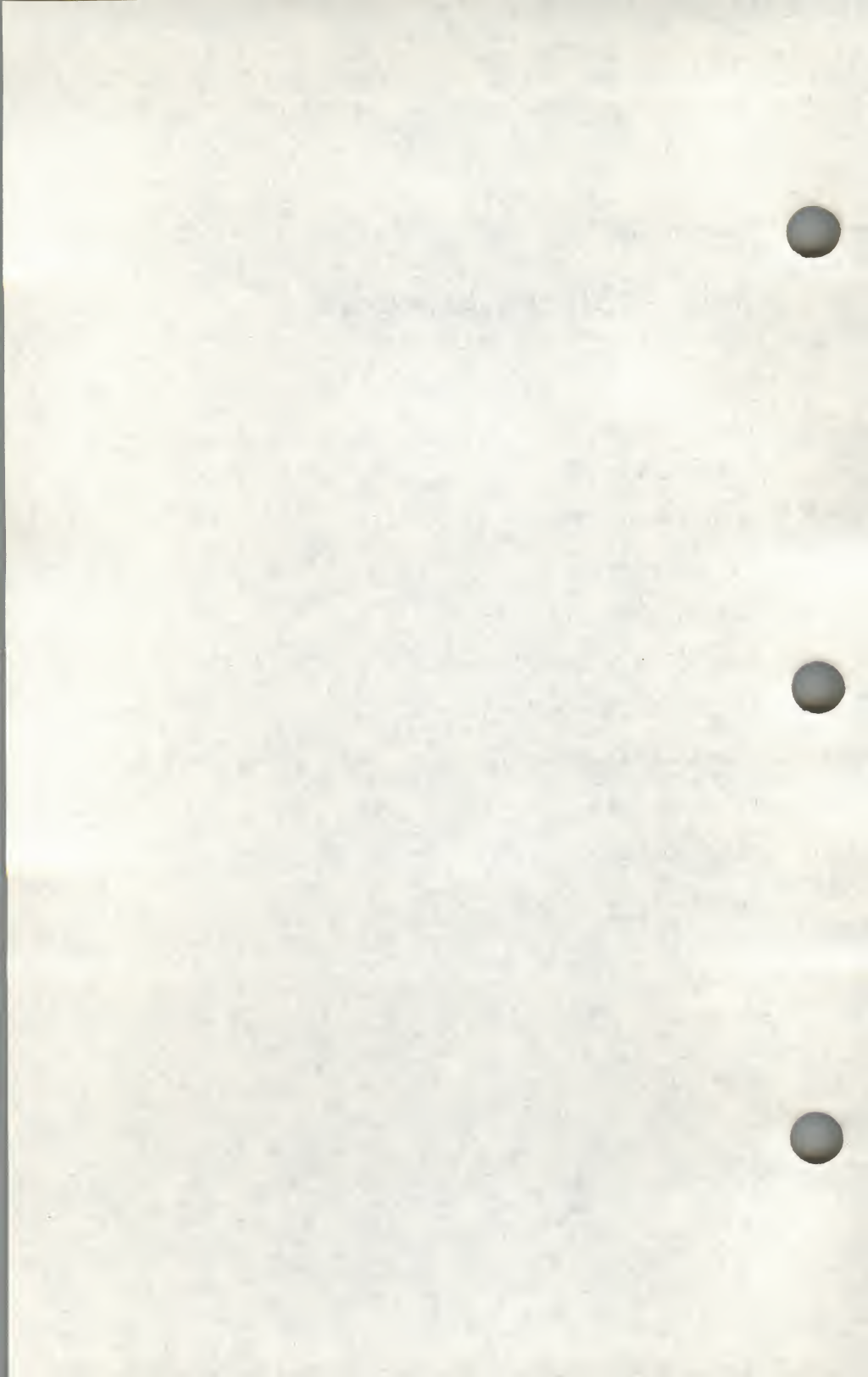
Manipulating Files 8-10

Using System Commands 8-11

Using Conditionals 8-12

Manipulating Strings 8-13

Printing 8-15



Introduction

The **m4** macro processor defines and processes specially defined strings of characters called macros. You can use the **m4** macro processor to enhance your programming language by defining a set of macros to be processed by **m4** and then using these macros in your programs. You can supplement your programming language with these macros to make your program more structured, readable, or appropriate for a particular application.

The major function of the **m4** macro processor is to replace one string of text with another as is done by the **#define** statement in C or the **define** construct in the **ratfor(CP)** command. Besides the straightforward replacement of one string of text with another, **m4** also provides:

- macros with arguments
- conditional macro expansions
- arithmetic expressions
- file-manipulation facilities
- string-processing functions

The basic operation of **m4** is to copy its input to its output. As the input is read, each alphanumeric string (that is, string of letters and digits) is checked. If the string is the name of a macro, the name of the macro is replaced by its defining text. The resulting string is reread by **m4**. Macros can also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

The **m4** macro processor provides a collection of about twenty built-in macros. In addition, the user can define new macros. This chapter describes some of the most commonly used built-in macros and explains how you can define your own macros. Built-in and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process. For more information about the built-in macros, see **m4(CP)** in the *UNIX Programmer's Reference*.

Invoking m4

To invoke **m4**, use a command of the form:

```
m4 [filenames]
```

Filename arguments are processed in order. If there are no arguments, or if an argument is a dash (-), then the standard input is read. The processed text is written to the standard output, and can be redirected as shown by the following command:

```
m4 file1 file2 - >outputfile
```

Note the use of the dash in the above example to indicate processing of the standard input after *file1* and *file2* have been processed by **m4**.

Defining Macros

The primary built-in function of **m4** is **define**, which is used to define new macros. The following statement defines the *name* string as *stuff*:

```
define(name, stuff)
```

All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter. (The underscore (`_`) counts as a letter.) The term *stuff* means any text, including text that contains balanced parentheses; it may stretch over multiple lines. The following example defines *N* to be 100 and uses this symbolic constant in a later **if** statement:

```
define (N, 100)
.
.
.
if (i > N)
```

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis it is assumed to have no arguments. This is the situation for *N*, since it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics, as shown in the following statements:

```
define (N, 100)
...
if (NNN > 100)
```

The *NNN* variable is absolutely unrelated to the defined macro *N*, even though it contains three *N*'s.

Defining Macros

Macro names or arguments can also be defined in terms of other names or arguments. The following statements define M and N each to be 100:

```
define(N, 100)
define(M, N)
```

In **m4**, if M is defined as N or as 100, M is 100. Therefore, even if N subsequently changes, M does not.

This behavior arises because **m4** expands macro names into their defining text as soon as it possibly can. This means that when the N string is seen, as the arguments of **define** are being collected, it is immediately replaced by 100. Therefore, you could have used the following statement in the first place:

```
define(M, 100)
```

If this isn't what you want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions as shown in the statements below:

```
define(M, N)
define(N, 100)
```

Now M is defined as the string N, so when you ask for M later, you will always get the value of N at that time (because the M will be replaced by N, which, in turn, will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by a grave accent and a single quotation mark ('and') is not expanded immediately, but has the marks stripped off. The following statements remove the punctuation marks from the N as the argument is being collected, but they have served their purpose, and M is defined as the N string, not as 100:

```
define(N, 100)
define(M, 'N')
```

The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word *define* to appear in the output, you have to quote it in the input, as shown below:

```
'define' = 1;
```

As another similar instance, consider redefining N with the following statements:

```
define(N, 100)
...
define(N, 200)
```

The N in the second definition is evaluated as soon as it is seen; that is, it is replaced by 100, which is the same as the following statement:

```
define(100, 200)
```

This statement is ignored by **m4**, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting as shown below:

```
define(N, 100)
...
define('N', 200)
```

In **m4**, it is often wise to quote the first argument of a define statement.

Quoting

If the acute and grave marks (`'` and ```) are not convenient for some reason, you can change the marks with the built-in function **changequote**. For example, the following statement defines the new quotation marks to be the left and right brackets:

```
changequote([, ])
```

You can restore the original characters by typing:

```
changequote
```

There are two additional built-in functions that are related to **define**. The built-in function **undefine** removes the definition of some macro or built-in function. The following statement removes the definition of `N`:

```
undefine('N')
```

Built-in functions can be removed with **undefine**, as in the following statement:

```
undefine('define')
```

Note

Once you remove a built-in function, you cannot get it back.

The built-in function **ifdef** determines whether a macro is currently defined. For instance, suppose that either *xenix* or *unix* is defined according to a particular implementation of a program. To perform operations according to the system you are using, you could use the following statements:

```
ifdef('xenix', 'define(system,1)' )  
ifdef('unix', 'define(system,2)' )
```

8

Don't forget the punctuation marks in the previous example.

The **ifdef** function actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as shown in the following statement:

```
ifdef('xenix', on XENIX, not on XENIX)
```

Using Arguments

So far you have learned the simplest form of macro-processing, that is, replacing one string with another (fixed) string. User-defined macros can also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of $\$n$ will be replaced by the n th argument when the macro is actually used. Thus, the **bump** macro, shown here, generates code to increment its argument by 1:

```
define(bump, $1 = $1 + 1)
```

Therefore, calling the **bump** macro as shown below will cause x to become $x+1$:

```
bump(x)
```

A macro can have as many arguments as you want, but only the first nine, **\$1** to **\$9**, are accessible. (The macro name itself is **\$0**.) Arguments not supplied are replaced by null strings, so you can define a macro, **cat**, which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Therefore, the following statement equals the expression xyz :

```
cat(x, y, z)
```

The arguments **\$4** through **\$9** are null, since no corresponding arguments were provided.

Leading unquoted spaces, TABs, or Newlines that occur during argument collection are discarded. All other white space is retained. Therefore, the following statement defines a to be $b\ c$:

```
define(a, b c)
```

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. Therefore, in the statement below there are only two arguments; the second is literally (b,c) :

```
define(a, (b,c))
```

Of course, a bare comma or parenthesis can be inserted by quoting it.

Using Built-in Arithmetic Values

The **m4** processor provides two built-in functions for doing arithmetic on integers. The simplest is **incr**, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than *N*, use the following statements:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

Precedence	Operator	Name
1	+	Arithmetic Addition
	-	Arithmetic Negation
2	**	Exponential
	^	Exponential
3	*	Multiplication
	/	Division
	%	Remainder
4	+	Addition
	-	Subtraction
5	= =	Equal
	!=	Not Equal
	<	Less than
	<=	Less than or Equal to
	>	Greater than
	>=	Greater than or Equal to
6	!	Logical NOT
7	&	Logical AND
	&&	Logical AND
8		Logical OR
		Logical OR

You can use parentheses to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like `1>0`) is 1, and false is 0. The precision in **eval** depends on the implementation.

For example, suppose you want M to be $2*N+1$, you can use the following statements:

```
define(N, 3)
define(M, 'eval(2*N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple (for example, just a number); this usually gives the result you want and is good programming form.

Manipulating Files

You can include a new file in the input at any time by using the built-in function **include**. The following statement inserts the contents of *filename* in place of the **include** command:

```
include(filename)
```

The contents of the file are often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file.

It is a fatal error if **include** cannot access the file named. To get some control over this situation, you can use the alternate form **sinclude**. **Sinclude** (silent include) says nothing and continues if it can't access the file.

You can also divert the output of **m4** to temporary files during processing and output the collected material upon command. The **m4** macro processor maintains nine of these diversions, numbered 1 through 9. The following statement puts all subsequent output at the end of a temporary file referred to as *n*:

```
divert(n)
```

Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion using the following statement:

```
undivert
```

This brings back all diversions in numeric order, and **undivert** with arguments brings back selected diversions in a given order. The act of undiverting discards the diverted text, as does diverting into a diversion whose number is not between 0 and 9, inclusive. The value of **undivert** is not the diverted text. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** function returns the number of the currently active diversion. This is zero during normal processing.

Using System Commands

You can run any program in the local operating system with the built-in **syscmd** function. For example, the following statement runs the **date** command:

```
syscmd(date)
```

Normally, **syscmd** would be used to create a file for a subsequent **include** command.

To make unique filenames easily, the built-in function **maketemp** is provided with specifications identical to the system function **mktemp**. A string of "XXXXXX" in the argument is replaced by the process ID of the current process.

Using Conditionals

There is a built-in conditional called **ifelse** that enables you to perform arbitrary conditional testing. In the simplest form, the following statement compares the two strings *a* and *b*:

```
ifelse(a, b, c, d)
```

If *a* and *b* are identical, **ifelse** returns the string *c*; otherwise, it returns *d*. Therefore, you might define a macro called **compare**, which compares two strings and returns “yes” if they are the same or “no” if they are different, as shown:

```
define(compare, `ifelse($1, $2, yes, no)`)
```

Note the quotation marks, which prevent the premature evaluation of **ifelse**. If the fourth argument is missing, it is treated as empty. **Ifelse** can actually have any number of arguments, and thus provides a limited form of multiple-decision capability. For example, the following statement compares the *a* and *b* strings. If they match, the result is *c*, and if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*.

```
ifelse(a, b, c, d, e, f, g)
```

If the final argument is omitted, the result is null, so the following statement evaluates to *c* if *a* matches *b*, and to null otherwise.

```
ifelse(a, b, c)
```

Manipulating Strings

The built-in **len** function returns the length of the string that makes up its argument. The following statement will return a value of 6:

```
len(abcdef)
```

All characters within the parentheses are counted, so the following statement will return a value of 5:

```
len( (a,b) )
```

The built-in **substr** function produces substrings of strings. The following statement returns the substring of *s* that starts at position *i* (origin zero) and is *n* characters long:

```
substr(s,i,n)
```

If *n* is omitted, the rest of the string is returned, so the following statement will return the string "ow is the time":

```
substr('now is the time', 1)
```

If *i* or *n* is out of range, various things result. For example, the following statement returns the index (position) in *s1* where the string *s2* occurs, or -1 if it doesn't occur:

```
index(s1,s2)
```

As with the **substr** function, the origin for strings is 0.

The built-in **translit** command performs character transliteration. The following command modifies *s* by replacing any character found in *f* with the corresponding character of *t*:

```
translit(s,f,t)
```

The following statement replaces the vowels with the corresponding digits:

```
translit(s, aeiou, 12345)
```

If *t* is shorter than *f*, characters that don't have an entry in *t* are deleted, as a limiting case. If *t* is not present at all, characters from *f* are deleted from *s*. Therefore, the following statement deletes vowels from "s":

```
translit(s, aeiou)
```


Manipulating Strings

There is also a built-in function called **dnl** which deletes all characters that follow it up to and including the next Newline. It is useful for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you use any of the following statements, the Newline character at the end of each line is not part of the definition, so it is copied into the output where it may not be wanted:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

If you add **dnl** to each of these lines, the Newline characters will disappear.

The following statements illustrate another way to eliminate unwanted Newline characters:

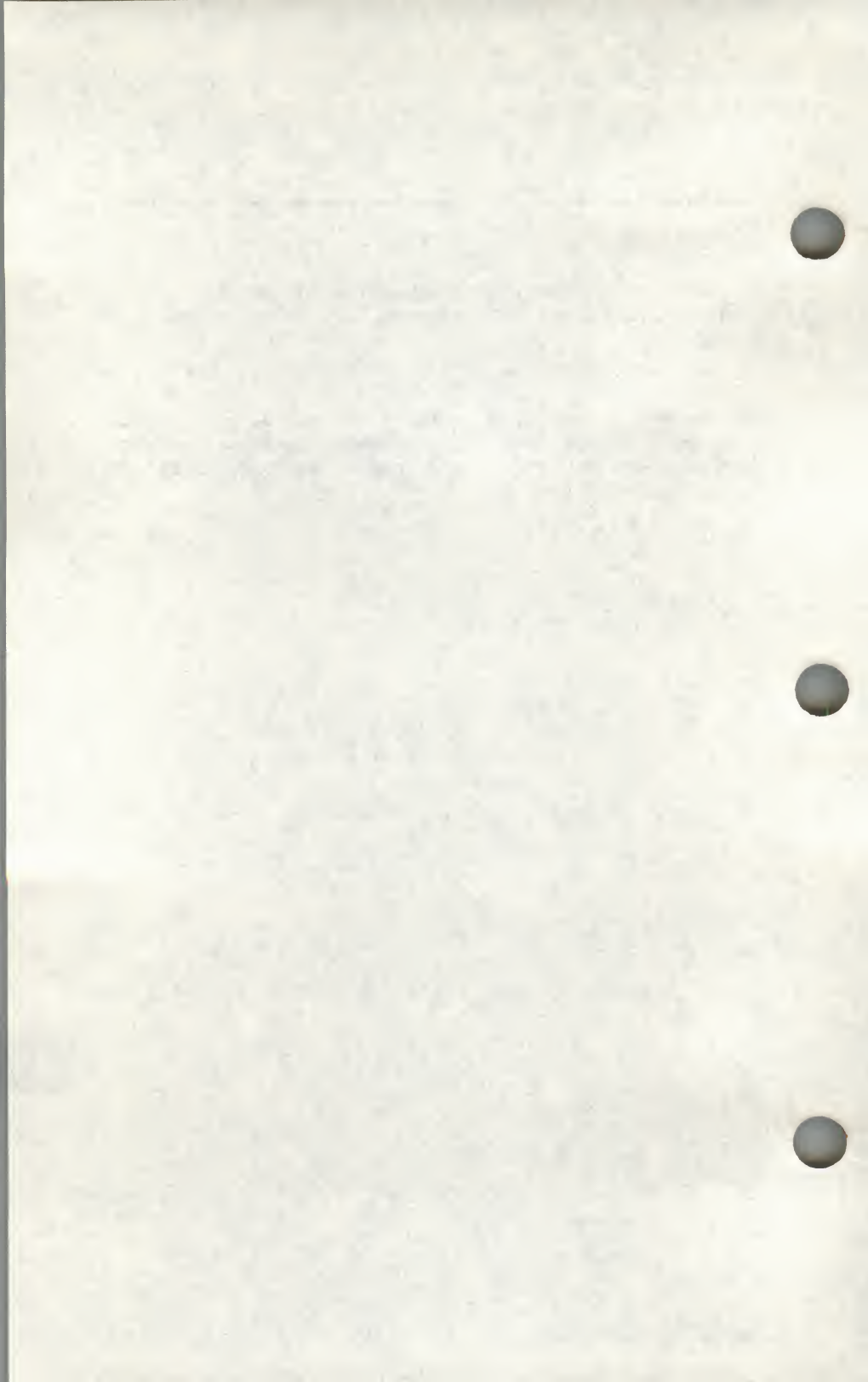
```
divert(-1)
  define(...)
  ...
divert
```

Printing

The built-in command **errprint** writes its arguments out on the standard error file. Thus, you can use the statement below to print a “fatal error” message:

```
errprint('fatal error')
```

The **dumpdef** function is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise, you get the ones you name as arguments. Don't forget the punctuation marks.



Chapter 9

C Programmer's Productivity Tools

Introducing the C Programmer's Productivity Tools 9-1

cscope 9-3

How **cscope** Works 9-3

The Cross-Reference File 9-5

Running **cscope** 9-6

Stacking **cscope** and Editor Calls 9-16

Directories Searched by **cscope** 9-16

Using Viewpaths 9-16

Changing a Text String 9-17

Changing a Constant to a Preprocessor Symbol 9-18

Adding an Argument to a Function 9-21

Changing the Value of a Variable 9-21

Cautionary Notes on Using **cscope** 9-22

Unknown Terminal Type 9-22

Dumping Core 9-22

Command Line Syntax for Editors 9-23

lprof 9-25

Introduction 9-25

Creating a Profiled Version of a Program 9-26

Running the Profiled Program 9-27

The PROFOPTS Environment Variable 9-27

Examples of Using PROFOPTS 9-28

Interpreting Profiling Output 9-31

Source Files in a Different Directory 9-34

Source Listing for a Subset of Files 9-34

Summary Option 9-34

Merging Option 9-35

Cautionary Notes on Using **lprof** 9-36

An Example of Using a Relative Path Name 9-37

Profiling Examples 9-40

Improving Performance with **prof** and **lprof** 9-40

Improving Test Coverage with **lprof** 9-47

the 11th of October 1891

My dear Mr. [Name]

I have just received your letter of the 10th

and am very glad to hear from you

I am well and hope this finds you the same

I have not much news to write at present

I am, dear Mr. [Name], very truly yours

[Signature]

[Address]

Introducing the C Programmer's Productivity Tools

This chapter will teach you how to use the C Programmer's Productivity Tools (CPPT). First, step-by-step instructions are provided through basic examples, so you can start using CPPT right away. Additional examples demonstrate various options that allow you to make the best use of the tools. To use CPPT, you must know how to use the other tools in the C Software Development Set.

The CPPT package consists of two tools: **cscope** and **lprof**; **cscope** is a browser; **lprof**, a profiler.

The **cscope** browser is an interactive program that locates specified parts of code in a set of C source files and gives you the option of editing those files. It can reduce significantly the amount of time you must spend searching for functions, function calls, macros, and variables in the code. Programmers responsible for writing programs (especially large ones) or maintaining existing programs will be able to edit their source code more efficiently with **cscope**. It is especially helpful for a programmer working on someone else's code. The section "**cscope**" is a tutorial on using this browser.

A profiler is a tool for analyzing a program's run-time behavior, a procedure known as dynamic analysis. The **lprof** tool allows a programmer or tester to identify those parts of the source code that are most often executed and those that are never executed when a program is run. The **lprof** tool provides line by line frequency profiling, reporting how many times each line of source code is executed. The **-x** option allows you to request test coverage analysis so that **lprof** only reports which lines of code are not actually executed at run time. It can be used over a set of tests such as are included in a test suite. The section "**lprof**" teaches you how to use this profiler to perform these types of dynamic analysis.

The section "Profiling Examples" presents examples of how profiling can be used to improve program performance. To enhance the effectiveness of **lprof**, use of another profiler, **prof**, is recommended. The **prof** tool reports the amount of time spent in various portions of a program. Once this has been determined, **lprof** can be used to obtain line-specific information about the heavily-executed portions of code identified by **prof**. These lines can then be rewritten to execute more efficiently.

Introducing the C Programmer's Productivity Tools

The **lprof** tool performs the following functions:

- produces source listings
- produces summary reports of profile data
- merges profile data files

cscope

How cscope Works

Imagine you arrive at work one day and are asked to learn how a particular program works. You are given a large stack of source code printouts and a cross-reference table for them. How do you go about studying the code? Until now, programmers have had to flip back and forth through pages of printouts to find the functions, function calls, macros, and variables listed in the cross-reference.

Now, however, you can use an interactive electronic tool to search through the code for you. This tool is the **cscope** browser. It builds a cross-reference symbol table for the functions, function calls, macros, and variables in the source files you designate. It then allows you to query that table about the locations of symbols you specify. Specifically, **cscope** presents a menu and asks you to choose the type of search you would like to have performed. For example, you may want **cscope** to find all functions that call a particular function.

When **cscope** has completed this search, it prints a list of the lines on which it has found the item (such as the functions calling a function) that you specified. It then waits for you to indicate which of these lines you want to examine. After you have requested a subset of the lines, **cscope** waits for you to edit a line (by using the default editor **vi** or an editor of your choice) or to begin another search.

Throughout a **cscope** session, you have the option of returning to the menu from the editor to request a new search. There is a variety of single-character commands available for manipulating the menu.

Because the procedure you follow will depend on the task you select, there is no single set of instructions for using **cscope**. To learn how this browser works, study the following example. It shows how you can locate a bug in a program without learning all the code.

Step 1: Identify the Problem

Suppose you are responsible for maintaining **cscope** itself. You notice that an error message, out of storage, sometimes appears when you run the program. How can you fix this? First, locate the parts of the code that are generating the message. Use **cscope** to find these parts quickly.

Step 2: Set Up the Environment

The **cscope** tool uses an editor as the medium you use to browse through your files. Therefore, before installing CPPT, you must check the environment to be sure an editor that can be used on your terminal is accessible.

Check the value of the **TERM** environment variable to make sure you have set it for your terminal. The first time you logged in you should have done this by assigning a value to **TERM** and exporting **TERM** to the shell, as follows:

```
$ TERM=term_name  
$ export TERM
```

You may now want to assign a value to the **EDITOR** environment variable. By default, **cscope** invokes the **vi** editor. If you prefer not to use **vi**, set the **EDITOR** environment variable to the editor of your choice and export **EDITOR**. (See the "Command Line Syntax for Editors" section under "Cautionary Notes on using **cscope**" for details and examples.)

If you want to use **cscope** only for browsing (without editing) you can set the **VIEWER** environment variable to **pg** and export **VIEWER**. The **cscope** tool will then invoke **pg** instead of **vi**.

Note

Using the **ed** or **jim** editor is possible but not recommended. (The **jim** editor takes advantage of the multi-screen capability of the AT&T 5620 terminal. It cannot be used with any other type of terminal.) See "Cautionary Notes on Using **cscope**" for suggested workarounds for these editors.

Once you have set up the environment so that **cscope** will call the editor of your choice, you are ready to use the browser.

Step 3: Invoke cscope

If all the source files for the program to be browsed (with the possible exception of standard system header files) are in the current directory, invoke **cscope** without any arguments:

```
$ cscope
```

By default, **cscope** builds its cross-reference table for all the C, **lex**, and **yacc** source files in the current directory. Therefore, typing **cscope** without any arguments is equivalent to the following command line:

```
$ cscope *.[chly]
```

The **cscope** tool will also search the standard directories for any header files that you include with **#include**.

Note

For other ways to invoke **cscope** (including a way to invoke it if the source files are in multiple directories), see “Other Command Line Options” later in this section.

The Cross-Reference File

When **cscope** is first invoked, it builds a cross-reference symbol table to which it refers during subsequent sessions. This table is created in the current directory and is called **cscope.out**. The next time **cscope** is invoked, it checks **cscope.out** for changes. The **cscope** tool modifies the table if the list of source files has been changed. Also, if the table has been modified, **cscope** rebuilds only the modified portions. Because copying information is much faster than building it, subsequent calls to **cscope** should require much less start-up time than the initial call.

cscope

Running cscope

After **cscope** has been invoked and the cross-reference information processed, the **cscope** menu of tasks appears on the screen:

Figure 9-1 The cscope Menu of Tasks

```
cscope          Press the ? key for help

List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

Press the **<TAB>** or **<Return>** key to move the cursor down the screen (with wraparound at the bottom of the display), and **<CTL>p** to move the cursor up. Once the cursor is at the desired input field, enter the text to be searched, and press the **<Return>** key.

The following single-key commands are available at any time during a **cscope** session.

Note

Instructions to type control characters (such as **<CTL>p** in the previous paragraph) should be followed by holding down the **<CTL>** key and pressing the letter shown.

Figure 9-2 Menu Manipulation Commands

<TAB>	move to next input field
<Return>	move to next input field
<CTL>m	move to next input field
<CTL>p	move to previous input field
.	search with the last text typed
<CTL>r	rebuild the cross-reference
!	start an interactive shell (type <CTL>d to return to cscope)
<CTL>l	redraw the screen
?	display list of commands
<CTL>d	exit cscope

Step 4: Locate the Source of the Error Message

Now let's return to the task we undertook at the beginning of the section **cscope**: to fix the problem that is causing the error message out of storage to be printed. You have invoked **cscope**; the menu is on the screen. Start your search for the problem by locating the section of code where the error message is generated. Move the cursor to the fifth menu item (List lines containing this text string) and enter the text **out of storage**.

Figure 9-3 Requesting a Search for a Text String

```
cscope          Press the ? key for help

List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string: out of storage
Change this text string:
List file names containing this text string:
```

Press the <Return> key. The **cscope** tool searches for the specified text and finds one line that contains it.

Note

Follow the same procedure to perform any other task listed in the menu, except the sixth, Change this text string. Because this task is slightly more complex than the others, there is a different procedure for performing it. For a description and examples of changing a text string, see “Examples of Using **cscope**” later in this section.

The **cscope** tool reports its finding as follows:

Figure 9-4 cscope Lists Lines Containing the Text String

Text string: out of storage

```

File      Line
1 alloc.c 56 (void) fprintf(stderr,
               "\n%s: out of storage\n",
               argv0 );

```

Edit this function or #define:
 List functions called by this function:
 List functions calling this function:
 List lines containing this text string:
 Change this text string:
 List file names containing this text string:

After **cscope** shows the results of a successful search in this way, you have several options. For example, you may want to edit one of the lines found. Or, if **cscope** has found so many lines that a list of them will not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after **cscope** has found the specified text.

Figure 9-5 Commands for Use after Initial Search

1-9	edit this line (the number you type corresponds to an item in the list of lines printed by cscope)
space	display the lines after the current line
+	display the lines after the current line
-	display the lines before the current line
<CTL>e	edit all lines
>	append the list of lines being displayed to a file

If the first character of the text for which you are searching matches one of these commands, be sure to precede it with a \ (backslash).

Now examine the code around the newly found line. Enter **1** (the number of the line in the list). The editor will be invoked with the file **alloc.c**; the cursor will be at the beginning of line 56 of the text file.

Figure 9-6 Examining a Line of Code Found by cscope

```

{
    return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char      *p;
{
    if (p == NULL) {
        (void) fprintf(stderr,
                        "\n%s: out of storage\n",
                        argv0);
        exit(C);
    }
    return(p);
}
~
~
~
~
~
~
"alloc.c" 60 lines, 1022 characters

```

By examining the code, you notice that the error message is generated when the variable *p* is NULL. To determine how an argument passed to **alloctest** could have been NULL, you must first identify the functions that call **alloctest**.

Exit the editor by using normal write and quit conventions, and return to the menu of tasks. Now type **alloctest** after the fourth item, List functions calling this function.

Figure 9-7 Requesting a List of Functions that Call **alloctest**

Text string: out of storage

```

File      Line
1 alloc.c 56 (void) fprintf(stderr,
                  "\n%s: out of storage\n",
                  argv0 );

```

List references to this C symbol:
 Edit this function or #define:
 List functions called by this function:
 List functions calling this function: **alloctest**
 List lines containing this text string:
 Change this text string:
 List file names containing this text string:

The **cscope** tool finds and lists three such functions.

Figure 9-8 **cscope** Lists Functions that Call **alloctest**

Functions calling this function: **alloctest**

```

File      Function  Line
1 alloc.c mymalloc  26 return(alloc(alloc(alloc((unsigned) size)));
2 alloc.c mycalloc  36 return(alloc(alloc(alloc((unsigned) nelelem,
                  (unsigned)
                  size)));
3 alloc.c myrealloc 46 return(alloc(alloc(alloc((unsigned)
                  size)));

```

List references to this C symbol:
 Edit this function or #define:
 List functions called by this function:
 List functions calling this function:
 List lines containing this text string:
 Change this text string:
 List file names containing this text string:

Now you want to know which functions call **mymalloc**. The **cscope** tool finds ten such functions. It lists seven of them on the screen and instructs you to press the space bar to see the rest of the list.

Figure 9-9 cscope Lists Functions that Call mymalloc

Functions calling this function: mymalloc

File	Function	Line
1 alloc.c	stralloc	17 return(strcpy(mymalloc(strlen(s) + 1), s));
2 dir.c	makesrcdirlist	70 srcdirs = (char **) mymalloc(nsrcdirs * sizeof(char *));
3 dir.c	makesrcdirlist	89 s = mymalloc(strlen(srcdirs[i]) + n);
4 dir.c	makefilelist	115 srcfiles = (char **) mymalloc(msrcfiles * sizeof(char *));
5 dir.c	makefilelist	116 srcnames = (char **) mymalloc(msrcfiles * sizeof(char *));
6 dir.c	addincdir	212 incdirs = (char **) mymalloc(sizeof(char *));
7 display.c dispinit		76 displine = (int *) mymalloc(mdisprefs * sizeof(int));

* 3 more lines - press the space bar to display more *

List references to this C symbol:

Edit this function or #define:

List functions called by this function:

List functions calling this function:

List lines containing this text string:

Change this text string:

List file names containing this text string:

Because you know that the error message (out of storage) is generated at the beginning of the program, you can guess that the problem may have occurred in the function **dispinit** (display initialization). To view **dispinit**, the seventh function on the list, type 7:

Figure 9-10 Viewing **dispinit** in the Editor

```

void
dispinit()
{
    /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
        mdisprefs = 9;
    }
    /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char    file[PATHLEN + 1];    /* file name */
    char    function[PATLEN + 1]; /* function name */
    char    linenum[NUMLEN + 1];  /* line number */
    int     screenline;           /* screen line number */
    int     width;                /* source line display
                                   width */

    register int    i, j;
    "display.c" 440 lines, 10198 characters

```

The **mymalloc** failed because it was called with either a very large number or a negative number. By examining the possible values of **FLDLINE** and **REFLINE**, you can see that there are situations in which the value of the variable is negative, that is, in which you are trying to call **mymalloc** with a negative number. The program needs a mechanism so that if the value of the variable is negative, it will abort after printing a meaningful error message.

For example, on an AT&T 5620 terminal, you may have multiple windows of arbitrary size. The error message might appear as a result of running **cscope** in a layer that has too few lines. One solution to this problem is to have the program print an error message stating that the screen is too small. Edit the function **dispinit** as follows:

cscope

Figure 9-11 Using **cscope** to Fix the Problem

```
/* initialize display parameters */

void
dispinit()
{
    /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs <= 0) {
        (void) fprintf(stderr, "\n%s: screen too small\n",
            argv0);
        exit(C);
    }
    if (mdisprefs > 9) {
        mdisprefs = 9;
    }
    /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}

~L/* display a page of the references */

void
display()
```

You have now corrected the problem we began investigating at the beginning of this section. If the screen is not large enough when you run your program in the future, the program will not simply fail with the cryptic error message out of storage. Instead, it will check the screen size and generate a more meaningful error message before exiting.

Other Command Line Options

The **cscope** tool examines all the **C**, **lex**, and **yacc** source files in the current directory by default. Thus:

```
$ cscope
```

is equivalent to:

```
$ cscope *.chly
```

The **cscope** command line provides several options that allow the programmer greater flexibility in selecting source files to be included in the cross-reference. To browse through specific files, invoke **cscope** with file names as arguments on the command line:

```
$ cscope file1.c file2.c file3.h
```

To specify a file containing a list of all the files to be browsed, use the **-i** option. If the source is in a directory tree, the following commands will allow you to examine all the source files easily:

```
$ find . -name '*.[chly]' -print | sort > filelist
$ cscope -i filelist
```

The **-I** option for **cscope** is similar to the **-I** option for **cc**. It directs **cscope** to search specified directories for **#include** files.

```
$ cscope -I./hdr
```

The **cscope** tool automatically searches for the **#include** files that it encounters when scanning.

The programmer can specify a cross-reference file other than **cscope.out** by using the **-f** option. This is useful for keeping separate symbol cross-reference files in the same directory. A programmer may want this if two programs are in the same directory but do not share all the same files.

```
$ cscope -f admin.ref admin.c common.c aux.c libs.c
$ cscope -f delta.ref delta.c common.c aux.c libs.c
```

In the preceding example, the source for two programs (**admin** and **delta**) are in the same directory, but the programs comprise different files. Suppose you are running **cscope** on **admin**. You may not want to see references to symbols in **delta.c**. By specifying two reference files, the cross-reference information for the two programs can be kept separate.

As with **cscope.out**, if the alternate file does not exist, **cscope** will build the cross-reference and leave it in the file specified by the **-f** option.

The **cscope** tool offers an option, **-d**, that allows you to prevent updating of the cross-reference table and thereby save time. It is permissible to use this option if you are sure that your source files have not been changed. However, because it is usually more important to safeguard against generating erroneous data than to save time, avoid using the **-d** option unless absolutely necessary.

Note

Use the **-d** option with extreme caution. If you specify **-d** with **cscope** under the erroneous impression that your source files have not been changed, **cscope** will give you data for an outdated program.

Optional Features

This section describes some of the more advanced capabilities of **cscope**.

Stacking cscope and Editor Calls

The **cscope** tool and editor calls can be stacked. This means that when **cscope** puts you in the editor to display one symbol reference and there is another symbol of interest, you can call **cscope** again from within the editor without exiting the current invocation of either **cscope** or the editor. You can then back up to a previous invocation by exiting the appropriate **cscope** and editor calls.

Directories Searched by cscope

The **cscope** tool searches for header files in directories in this order:

1. the current directory
2. directories specified by the **-I** option (if they exist)
3. the standard location for header files (usually **usr/include**).

The **cscope** tool searches for source files only in the current directory.

Using Viewpaths

The environment variable **VPATH** replaces the current directory in the order of directories searched by **cscope**. This enables you to extend your search for source files from a single directory to a set of directories.

Note

You must specify your current directory in **VPATH** if you want it to be searched. (The current directory can always be represented by the **.** symbol.)

To set **VPATH**, list the directories you want searched (by their path names) in the order you want them searched. Separate each directory name with colons.

For example, suppose you have a program that consists of three files, **a.c**, **b.c**, and **c.c**. You have assigned the following directories to the **VPATH** variable:

```
$ VPATH=/fs2/mydirectory:/fs1/delivered:/fs1/proj/official
```

First **cscope** searches for the file **a.c** in the directory **/fs1/mydirectory**. If the file is not in that directory, **cscope** continues searching for it in the other directories specified in **VPATH** until it finds the file. Similarly, **cscope** searches the directories in the specified order for **b.c** and **c.c**.

Examples of Using cscope

This section presents examples of how **cscope** can be used to perform three tasks: change a constant to a preprocessor symbol, add an argument to a function, and change the value of a variable.

Changing a Text String

The standard procedure for calling tasks listed on the **cscope** menu of tasks was described in “Running **cscope**” under “Step 3: Invoke **cscope**” early in this section. One task on the menu differs slightly from the others and necessitates following a different procedure.

If you select the sixth menu item, **Change this text string**, **cscope** will prompt you for new text and then display the lines containing the old text. You can select the lines you want changed with any of the following single-key commands:

Figure 9-12 Commands for Selecting Lines to be Changed

1-9	mark or unmark the line to be changed
*	mark or unmark all displayed lines to be changed
space	display next lines
+	display next lines
-	display previous lines
a	mark all lines to be changed
<CTL>d	change the marked lines and exit
<ESC>	exit without changing the marked lines

The rest of this section consists of more detailed examples.

Changing a Constant to a Preprocessor Symbol

Suppose you want to change a constant, 100, to a preprocessor symbol, MAXSIZE. Select the sixth menu item (Change this text string) and enter **\100**.

Note

The **1** must be preceded by a **** (backslash) because it has a special meaning (item 1 on the menu) to **cscope**.

Now press **<Return>**; **cscope** will prompt you for the new text string. Type **MAXSIZE**.

Figure 9-13 Changing a Text String

cscope

Press the ? key for help

```

List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string: 100
List file names containing this text string:
To: MAXSIZE

```

The **cscope** tools then displays the lines containing the particular text string, and waits for you to specify the subset of these lines in which you want the text to be changed.

Figure 9-14 cscope Prompts for Lines to be Changed

Change "100" to "MAXSIZE"

```

File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

```

```

List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
Select lines to change (press the ? key for help):

```

You know that occurrences of 100 in lines 1, 2, and 3 of the list (from lines 4, 26, and 8 of the program) are to be changed to MAXSIZE. However, the occurrences of 100 in **read.c** and **err.c** (lines 4 and 5 of the list) are not related; in these lines, 100 should not be changed. Enter 1, 2, and 3.

The numbers you type are not printed on the screen. Instead, **cscope** prints a > (greater than) symbol after each number of the list that you type. For example, after you type 1, a > symbol is printed after the number 1 (and before the line **init.c 4 char s[100];**) in the list, as shown in Figure 9-15.

Figure 9-15 Marking Lines to Be Changed

Change "100" to "MAXSIZE"

```
File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
Select lines to change (press the ? key for help):
```

After selecting lines, type <CTL>d to change them. Then **cscope** displays the lines that have been changed.

Figure 9-16 cscope Displays Changed Lines of Text

Changed lines:

```
char s[MAXSIZE];
for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {
```

Type any character to continue:

When you type a character in response to this prompt, **cscope** will pause and redraw the screen before allowing you to continue with the session, as shown in Figure 9-17.

The next step is to add the **#define** for the new symbol **MAXSIZE**. Escape to the shell by typing **!**. (The shell prompt will appear at the bottom of the screen.) Then enter the editor and add the **#define**.

Figure 9-17 Escaping from **cscope** to the Shell

```
Text string: 100
```

```
File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
$ vi defs.h
```

To resume the **cscope** session, quit the editor and type **<CTL>d** to exit the shell.

Adding an Argument to a Function

The **cscope** tool makes it easy to add an argument to a function. This involves two steps: editing the function itself and adding the new argument to each place where the function is called.

First, edit the function by using the second menu item, **Edit this function or #define**. Next, find out where the function is called. By invoking the fourth menu item, **List functions calling this function**, you can get a list of all functions that call it. With this list, you can either invoke the editor on each line found by entering the list number for each line individually, or invoke the editor on all lines automatically by typing **<CTL>e**. To make this kind of change, **cscope** is especially useful because it guarantees that none of the functions you need to edit will be overlooked.

Changing the Value of a Variable

The value of **cscope** as a browser becomes apparent when you want to see how a proposed change will affect your code. Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item (**List references to this C symbol**) to obtain a list of references that will be affected. Then use the editor to examine each one. This will help you predict the overall effects of your proposed change. Later, you can use **cscope** this way again to verify that your changes have been made.

Cautionary Notes on Using cscope

This section describes solutions for several problems that may arise while you are using **cscope**.

Unknown Terminal Type

You may see the following error message:

```
cscope: "term" is not in the terminal database.
```

If this message appears, your terminal may not be listed in the terminal information (terminfo) database that is currently loaded. Try reloading the database from the Terminal Information Utilities.

You may also see:

```
cscope: TERM variable is not set or is not exported in your .profile
```

If this message appears, set and export the TERM variable as described at the beginning of this section. (See “Step 2: Set Up the Environment.”)

Dumping Core

Your system may dump core if the following sequence of events occurs:

1. You make changes to your source code using **cscope**.
2. You rebuild the cross-reference table using the **<CTL>R** command. (**<CTL>R** rebuilds the table only if you have made changes.)
3. After the table has been rebuilt, the list of references previously displayed becomes obsolete. The screen is cleared so it resembles the initial screen shown by **cscope**. (See Figure 9-1.)
4. If you try to append the contents of this screen (nothing) to a file by using the **>** command, **cscope** will dump core and leave your terminal in an unusable state.

To avoid this situation, make sure you see lines of text displayed before trying to append them to a file.

Command Line Syntax for Editors

By default, **cscope** invokes the **vi** editor. The **cscope** tool expects **vi** and any other editor it uses to have a standard command-line syntax of the following form:

editor +linenum filename

If you want to use an editor that has this command-line syntax, set the **EDITOR** environment variable to the editor of your choice and export **EDITOR**. For example, if you want to use the **emacs** editor, enter the following commands:

```
$ EDITOR=emacs
$ export EDITOR
```

However, if the editor you want to use does not conform to this command-line syntax, you must write an interface between **cscope** and the editor.

For example, suppose you want to use **ed**. You have already set the **EDITOR** variable to **ed** and exported it. However, because the **ed** editor does not allow specification of a line number on the command line, you will not be able to edit or view any files while using **cscope**. To solve this problem, write a shell script called **myedit** that contains the following line:

```
/bin/ed $2
```

Then set the value of **EDITOR** to your shell script:

```
$ EDITOR=myedit
```

Now, when **cscope** invokes the editor, it calls this shell script with the following command line:

```
myedit +17 main.c
```

The **myedit** tool will discard the line number (\$1) and call **ed** correctly with the filename (\$2).

Note

The **ed** tool has one other drawback as a **cscope** editor that you should take into consideration when selecting an editor: it cannot move you to specified lines in the file. If you use the shell script shown in the previous example, you will have to move to specified lines manually.

lprof

Introduction

As described in "Introducing the C Programmer's Productivity Tools," there are two profilers available for dynamic analysis of C programs written in a UNIX System environment.

- The **prof** tool performs time-profiling; it reports how much time is spent executing various portions of a program.
- The **lprof** tool performs line-by-line frequency-profiling; it reports how many times each line of source code is executed.

To use either of these profilers, you must follow a three-step procedure.

Step 1: Compile your program with a profiling option.

for **prof**: **cc -qp** (or **-p**)
for **lprof**: **cc -ql**

Step 2: Run the profiled program so that run-time data can be collected. At the end of execution, the run-time data is written to another file, known as a data file. This consists of a header section, followed by a section for each function and an end-of-data marker at the end of the file. The coverage data (execution count) for each function is recorded alongside the function's name.

Data files have the following default names:

for **prof**: **mon.out**
for **lprof**: **prog.cnt**

where *prog* is the name of the profiled program.

Step 3: Examine the data by running a profiler with the **prof** or **lprof** command.

lprof

The following three sections explain these steps in detail; providing an example of how to perform dynamic analysis of a file called **travel.c**.

Creating a Profiled Version of a Program

What must you do to profile a file with **lprof**? Suppose you have a file called **travel.c**. (This is a hypothetical example; CPPT does not include such a file.) Start by creating an executable file (**a.out**) from the source file (**travel.c**). Use the **-ql** option with the **cc** command so that line count data will be saved:

```
$ cc -ql travel.c
```

If you want to use a **cc -c** command line, you must specify **-ql** when you link as well as when you compile:

```
$ cc -ql -c travel.c
$ cc -ql -c misc.c
$ cc -ql -o travel travel.o misc.o
```

These sample command lines illustrate what you must do to profile an entire program. However, you may be interested in profiling only a piece of a large program. To profile an individual source file, create a profiled version in the same way: specify the **-ql** option with the **cc** command, both when you compile and when you link the files.

For example, suppose you have a program composed of two source files: **travel.c** and **misc.c**. By running **prof** on both files, you find out that 70% of the total execution time can be accounted for by one function in **travel.c**. You now want to examine that function with **lprof** to determine how you can improve its performance. Run the **cc** command with the **-ql** option on the **travel.c** file alone and again when you link **travel.c** and **misc.c**.

```
$ cc -ql -c travel.c
$ cc -c misc.c
$ cc -ql -o travel travel.o misc.o
```

The final result will be a program called **travel**.

Running the Profiled Program

Now execute **travel** so that run-time data can be collected. This information is stored in a data file called **travel.cnt** in your current directory. When the program ends, the following message is printed to **stderr**:

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' created
```

This is how **lprof** handles run-time data by default. However, if you prefer, you can specify how you want this data to be handled by setting options for an environment variable called **PROFOPTS**.

The PROFOPTS Environment Variable

The environment variable **PROFOPTS** provides run-time control over profiling. When the profiled program is about to terminate, it examines the value of **PROFOPTS** to determine how the profiling data is to be handled.

The **PROFOPTS** environment variable is a comma-separated list of options interpreted by the program being profiled. If **PROFOPTS** is not defined in the environment, then the default action is taken: the profiling data is saved in a file (with the default name) in the current directory. If **PROFOPTS** is set to the null string, no profiling data is produced.

The following options can be specified for **PROFOPTS**. They are explained in more detail in the examples.

msg=[y/n] If **msg=y** is specified, print a message (to **stderr**) stating that profile data is being created. If **msg=n** is specified, print only profiling error messages. The default is **msg=y**.

merge=[y/n] If **merge=n** is specified, do not merge data files after successive runs; the data file will be overwritten after each execution. If **merge=y** is specified, the data will be merged. The merge will fail if the program has been recompiled; the data file will be stored in **TMPDIR**. The default is **merge=n**.

lprof

pid =[y / n]	If pid=y is specified, the name of the data file will include the process ID of the profiled program. This allows the creation of different data files for programs calling fork(S) . If pid=n is specified, the default name is used. The default is pid=n .
dir = <i>dirname</i>	Place the data file in the directory <i>dirname</i> , if this option is specified. Otherwise, the data file is created in the directory that is current at the end of execution.
file = <i>filename</i>	Use <i>filename</i> as the name of the data file in <i>dir</i> created by the profiled program, if this option is specified. Otherwise the default name is used. (See “Profiling Programs that Fork” for an example.)

Examples of Using PROFOPTS

The following sections provide examples of how PROFOPTS might be used, in typical profiling situations, to tailor the environment for specific tasks.

Turning Off Profiling

If you do not want to profile a particular run, you can set PROFOPTS to the null string on the command line when you run a profiled version of a program:

```
$ PROFOPTS="";
```

However, this value will remain in effect for only one execution of one program.

If you want to turn off profiling for more than one program and/or run, you must export the value of PROFOPTS:

```
$ PROFOPTS="" export PROFOPTS  
$ travel
```


Exporting the variable eliminates the need to specify it every time you run **travel**. It also makes the value of **PROFOPTS** applicable to all runs of any profiled programs, not just **travel**. Once you have exported **PROFOPTS**, it keeps the value you have given it until you **unset** or **redefine** that variable.

Merging Data Files

Suppose you are not interested in the data from a single run; you want the information collected from all runs. A data file containing information from multiple executions is called a merged data file. When data files created with the **lprof** compiling option are merged, the execution counts for all files are added together arithmetically.

The following screen shows how you must specify the environment if you want your data files from successive runs to be merged:

```
$ PROFOPTS="merge=y"
$ export PROFOPTS
$ travel
```

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' created
```

```
$ travel
```

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' updated
```

Keeping Data Files in a Separate Directory

To avoid clutter in your current directory, you may want to create a directory for data files. If you do, be sure to specify that directory on your command line. For example:

```
$ PROFOPTS="dir=cntfiles" travel
```

All the data files will be created in the subdirectory **cntfiles**.

Profiling within a Shell Script

You may want to write a shell script that runs profiled programs automatically. This could be useful for specific tasks that you perform frequently, such as determining coverage. For example:

- You might not want to receive notification (via a message sent to *stderr*) that profiling data is being created.
- You might want to have data merged automatically.
- You might want to give the data files names that you can associate with a specific test-case run.

You can specify these conditions by using **PROFOPTS** as follows:

```
$ PROFOPTS="msg=n, merge=y, file=test1.cnt" myprog < test1
```

Here, because all the data files in the directory are for the program **myprog**, the file name **test1.cnt** conveys more information than **myprog.cnt**.

Profiling Programs that Fork

If a program uses the system call **fork(S)**, the data files of the parent and child processes will have the same name by default. You can avoid this by using the **PROFOPTS** option **pid**. By setting **pid**, you ensure that the data file name will include the process ID of the program being profiled. As a result, multiple data files will be created, each with a unique name.

What happens when you run a program that forks without using the **pid** option? If you have set **merge=y**, the data will be merged; data from separate processes will be indistinguishable. If you have set **merge=n**, the last process to dump data will overwrite the data file.

The following screen shows how the **pid** option works. Notice the data files that are created (as reported by the messages sent to *stderr*) by the command line at the top of the screen:

```
$ PROFOPTS="pid=y" forkprog
```

```
dumping profiling data from process 'forkprog' . . .
      CNTFILE '922.forkprog.cnt' created
```

```
dumping profiling data from process 'forkprog' . . .
      CNTFILE '923.forkprog.cnt' created
```

Interpreting Profiling Output

You can use **lprof** to:

- produce source listing reports of profile data
- produce summary reports of profile data
- merge profile data files

Specifying a Program and Data File to lprof

The **lprof** tool interprets both a profiled program and the data file associated with it to produce profiling information. By default, **lprof** expects the profiled program to be called **a.out**, and the data file, **a.out.cnt**.

To run **lprof** on a program with a name other than **a.out**, specify the name after the **-o** option. For example, to run **lprof** on a program called **sample** use the following command line:

```
$ lprof -o sample
```

The **lprof** tool will assume that the data file is called **sample.cnt**.

You can also specify a data file other than **sample.cnt** by using the **-c** option.

```
$ lprof -c newdata.cnt
```

The name of the profiled program is stored, exactly as it appears on the command line, in the data file. (Because the **-o** option is not specified, the profiled program consults the data file to obtain the name of the program.) Therefore, the simplest way to invoke **lprof** is to specify the name of the data file and let **lprof** determine the name of the program. Because the name of the data file is not stored in the program itself, the reverse is not true: you cannot specify the name of the program and expect **lprof** to determine the name of the data file if it is not the default name.

Source Listing Option

Along with profiling information, **lprof** produces a source listing by default. Once you have executed your profiled program and the data file has been created, you can view the profile data by entering the following command:

```
$ lprof
```


The **lprof** output consists of a source listing with profiling information in the left margin, as shown in the following example:

Figure 9-18 Example of lprof Output

```

#include <stdio.h>

main()
1 [4]  {
        /* note that declarations are not
executable lines and therefore have no
line-number or execution status
associated with them */
        int i;

1 [11]      for (i = 0; i < 10; i++)
10 [12]      sub1();

1 [14]      }

        sub1()
10 [17]      {
        /* but here, this declaration
is an executable statement */
10 [19]      int i = 0;

10 [21]      if (i > 0) {
        /* next line is an example
of code never executed */
0 [23]      sub2();
        }
        else {
10 [26]      sub3();
        }

10 [28]      }

        sub2()
0 [31]      {
        /* do nothing */
0 [33]      }

        sub3()
10 [36]      {
        /* do nothing */
10 [38]      }

```

The square brackets enclose line numbers for the file. Each number to the left of a line number shows how many times the corresponding source line was executed.

If you use the **-x** option to **lprof**, the output highlights the lines that have not been executed. Lines that have been executed are marked only by line numbers. Lines that have not been executed are marked with a line number preceded by a **[U]**. Figure 9-19 shows an example of output produced by the **-x** option:

Figure 9-19 Example of Output Produced by the **-x** Option

```

#include <stdio.h>

main()
[4]  {
        /* note that declarations are not
        executable lines and therefore have no
        line-number or execution status
        associated with them */

        int i;

[11]     for (i = 0; i < 10; i++)
[12]     sub1();

[14]  }

sub1()
[17]  {
        /* but here, this declaration
        is an executable statement */
[19]     int i = 0;

[21]     if (i > 0) {
        /* next line is an example
        of code never executed */
[U] [23]     sub2();
        }
        else {
[26]     sub3();
        }

[28]  }

sub2()
[U] [31]  {
        /* do nothing */
[U] [33]  }

sub3()
[36]  {
        /* do nothing */
[38]  }

```

lprof

In any **lprof** output, certain lines (such as declarations, comments, and blank lines) do not have line numbers associated with them. This allows you to distinguish between lines that were not executed during a particular run from those that are not executable. In the previous example, neither line 22 nor line 23 in **sub1** was executed, but line 23 is marked with a line number while line 22 is not. This is because line 22 is not executable; line 23 is executable but was not executed in the run that produced this output.

Source Files in a Different Directory

The **lprof** tool assumes, by default, that the source files for the program you specify are in the current directory. If they are in another directory, you must specify their location with the **-I** option and a path name. For example, to specify source files in the **/usr/src/cmd** directory, use the following command line.

```
$ lprof -o cat -c cat.cnt -I /usr/src/cmd
```

In this line, **lprof -I** instructs **lprof** to search for the specified source file, **cat.c**, in the specified directory. You can specify multiple **-I** arguments on one command line.

Source Listing for a Subset of Files

If you want profiling output for a limited number of selected files, use the **-r** option with **lprof**:

```
$ lprof -r file1.c -r file2.c
```

This command line will produce output only for **file1.c** and **file2.c**. This is useful if you want to examine a few files rather than an entire program.

Summary Option

You can obtain a summary report of the profile data by specifying **lprof -s**:

```
$ lprof -s -c sample.cnt
```

Because a source listing is not produced with **lprof -s**, the **-r** and **-I** options do not need to be specified. The following screen shows an example of output produced with the **-s** option.

Figure 9-20 Example of lprof -s Output

```
Coverage Data Source: sample.cnt
Date of Coverage Data Source: Mon Apr 7 17:19:43 1986
Object: sample
```

percent covered	lines covered	total lines	function name
100.0	4	4	main
83.3	5	6	sub1
0.0	0	2	sub2
100.0	2	2	sub3
78.6	11	14	TOTAL

Merging Option

As described in the section on the PROFOPTS environment variable, data files can be merged automatically at run-time. You can also merge existing data files with the **lprof** command.

```
$ lprof -d destfile -m file1.cnt file2.cnt file3.cnt
```

The command line requires both **-d** and **-m**. The **-m** option takes the names of two or more data files to be merged. The **-d** option specifies the destination file (the new file) that will contain the merged data. The data files must have been created by the same profiled program; if they have not, **lprof** will issue an error message.

```
$ lprof -d merged.cnt -m prog1.cnt prog2.cnt
```

```
ERROR: 'prog1', 'prog2'
```

```
Object file entry names & timestamps don't match.
```

```
*** no merged output ***
```

However, you may have multiple data files, created by the same program, that have different time stamps. This will happen, for example, if you recompile a program. If you want to merge data from runs of different versions of the same program, you can override the time-stamp check by specifying **-T** (time stamp override).

Note

You must be extremely cautious when using the **-T** option. If the control flow of the recompiled program has changed, the new merged data file is very likely to be erroneous; **lprof** will produce an incorrect report.

Cautionary Notes on Using lprof

This section describes solutions for several problems that may arise while you are using **lprof**.

Trouble at Compile Time

On rare occasions, when compiling a file with the profiling option, you may receive a warning that a particular function is not being profiled:

```
$ cc -c -ql -O file.c
```

```
>>> BASICBLK WARNING - not profiling function fname: [trouble at line n]
```

The reason for this may be that you are using the optimizer together with the profiling option. This is usually a permissible combination of options, but occasionally the compiler does not accept it.

You may not need to have the function in question profiled. If not, ignore this warning; data will be collected in the data file for all other functions. If you do want data for the function in question, compile your program again with the profiling option but without optimization. The warning should not reappear.

Non-Terminating Programs

If the profiled program does not terminate, no profiling data will be saved. The profiling data is saved at termination by the system call **exit(S)**. If **exit(S)** is never called, no profiling data is saved.

Failure of Data to Merge

If a program has been recompiled, a new data file will be created in a temporary directory. The path name of the new file will be printed to *stderr*.

Specifying Program Names to lprof

When the profiled program is run, the name of that program is stored, exactly as it appears on the command line, in the data file. The simplest way to invoke **lprof** is to specify the name of the data file and let **lprof** determine the name of the program. However, because the name of the data file is not stored in the program itself, the reverse is not true: you cannot specify the name of the program and expect **lprof** to determine the name of the data file if that name is not the default name.

The **lprof** tool will not be able to display data if you do the following two steps in the order shown:

1. Use a relative path name on the command line when you run your profiled program.
2. Run **lprof** from a different directory, specifying only the name of the data file (that is, without specifying the program name).

When you run **lprof** from a directory other than the one in which you have executed your profiled program, and you have used a relative path name when executing the profiled program, then you must specify the **-o** option with either the profiled program's full pathname or the program's pathname relative to your current directory.

An Example of Using a Relative Path Name

Say you are working in a directory called *cur.dir*. You have compiled a program called *newprog.c* and gotten the profiled version, **newprog**. Now you execute **newprog**. A data file called *newprog.cnt* is created in your current directory (*cur.dir*). It includes the name of the profiled version you executed, in the form you entered it on the command line: **newprog**. After **newprog** has finished running, you change directory to **\$HOME**. Now you want to examine the results of the execution of **newprog**. From **\$HOME**, you enter the following command line:

```
lprof -c cur.dir/newprog.cnt
```


lprof

Because the data file has stored the name of the profiled file as you entered it on the command line (**newprog**), **lprof** now looks for (and fails to find) **newprog** in the current directory (**\$HOME**). You will receive an error message:

```
***cannot access object file 'newprog'***
```

Note

The term `object file` refers to the profiled version of your file.

To make sure that **lprof** can access the profiled file, specify its relative path (from **\$HOME**) with the **-o** option, as follows:

```
lprof -c cur.dir/newprog.cnt -o cur.dir/newprog
```

Trouble at the End of Execution

At the end of execution, you may see the following error message:

```
dumping profiling data from process 'a.out' . . .  
***unable to seek to symbol table
```

Usually this is caused by running a stripped version of a profiled program. Never strip files to be profiled. If necessary, change makefiles so that they do not produce stripped files.

No Data Are Collected

You may get no data after running a profiled program. The program terminates normally, and you receive neither a message about data being saved, nor an error message. This may be caused by one of two problems:

- You may not have specified **-ql** at both compile time and link time. If you forget to specify **-ql** when you link, the profiled program will run, but a data file will not be created.

- The profiled program may include a call to `_exit` that is causing the program to quit without calling `exit(S)`, the procedure that saves your profiling data. Replace calls to `_exit` with calls to `exit(S)` to save profiling data.
- The `PROFOPTS` variable may be set to `NULL`.

Data File Cannot Be Found

Occasionally, you may not be able to find the data file, despite the fact that the profiled program has terminated normally and you have received a message saying that the data file has been created.

The profiled program creates the data file in the directory in which the program is located when it terminates. If the program changes directories, the data file may be created in a directory different from both the one from which you executed the program and the one in which the shell is located when the program terminates.

Use the `dir` option of `PROFOPTS` to specify exactly where the data file is to be created, so you will be able to find it.

Using lprof with Shared Libraries

It is recommended that when profiling with **lprof**, you use archived versions of libraries rather than shared versions. If you must profile with a shared library (for example, if an archived version is unavailable), you need to specify all necessary options on the `ld(CP)` command line at link time. The `ld(CP)` command is documented in the *Programmer's Reference*.

After compiling as usual with the `-ql` option (as described earlier in this section), link by invoking `ld(CP)` directly, as follows:

```
$ ld user_opts /lib/pcrt1.o files.o -lprof -lld -lm -lc -lg /lib/crtn.o
```

The `user_opts` are options, such as `-o prog`, that you normally specify on the `cc(CP)` command line. The `cc(CP)` command is also in the *Programmer's Reference*.

You must also check any makefiles to be sure the `ld(CP)` command is invoked (instead of the `cc(CP)` command) and has the appropriate options, as shown in the preceding example command line. See the `ld(CP)` manual page for details about options available with `ld(CP)`.

Profiling Examples

Improving Performance with **prof** and **lprof**

The problem of how to improve program efficiency is addressed by Jon Bentley in *Writing Efficient Programs* (Englewood Cliffs: Prentice-Hall, 1982). Bentley observes that:

- A small part of the code usually accounts for a high percentage of the run time.
- Programmers have difficulty identifying the most time-consuming parts of the code.

To solve the second problem, he recommends the use of profilers. The **prof** and **lprof** profilers can help a programmer locate the time-consuming parts of a C program.

Specifically, **prof** provides a time profile, that is, a list of the most time consuming functions and the amount of time taken by each. The **lprof** tool provides a list of the lines that are being executed most frequently. Once these potential problem areas have been identified, it is the programmer's job to rewrite those parts of the code so that the program runs more efficiently.

Although either of these profilers can be used singly, they are most efficient if you use them together, as follows. First, profile your program with **prof** to identify the most time-consuming functions. Then, profile only those functions (rather than the entire program) with **lprof** to determine which lines are being executed most frequently.

This two-step approach takes the guesswork out of determining which lines of code are the most time-consuming. Bentley notes that although programmers want to save time by profiling only selected parts of their codes instead of whole programs, they seldom select the correct routines to monitor.

He also emphasizes the importance of profiling programs with data that are typical of what the program will encounter in normal use. Most test cases fail to provide profiling data representative of typical usage.

In the next section, an example will be detailed to illustrate how **prof** and **lprof** can be used to improve program performance.

lprof on lprof

During the development of **lprof**, it was observed that the process of merging profile data was slow. The profiling data being merged came from two runs of the C compiler, which is a medium-sized program with 284 functions. It took forty cpu seconds (two minutes of real time) to merge the two coverage files.

The first step was to produce a time profile of **lprof** to see which functions were taking the most time. Here is part of the output from **prof**:

Figure 9-21 **prof** Output

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
34.8	13.52	13.52	226638	0.0597	fread
12.1	4.72	18.24	228254	0.0207	memcpy
9.5	3.69	21.93	40286	0.0918	CAjump
9.2	3.60	5.52			_mcount
7.7	2.99	28.51	284	10.53	CAfind
7.6	2.94	31.45	42472	0.0692	malloc
6.3	2.45	33.90	1154	2.123	read
3.0	1.17	35.07	40475	0.0289	strcmp
2.8	1.09	36.16	42471	0.0257	free
2.5	0.96	37.13	2	482.	creat
1.4	0.55	37.68	1	550.	fputc
0.8	0.33	38.01	1431	0.231	lseek
0.4	0.16	38.17	1518	0.105	fwrite
0.3	0.11	38.28	569	0.19	Caread

The two most time-consuming user functions were **CAjump** and **CAfind**. We wondered why **CAjump** was called 40,286 times and why the average time per call for **CAfind** was so high (10.53 milliseconds).

Profiling Examples

The next step was to run **lprof** on these two functions. Here are the results of running **lprof** on the function **CAfind**:

Figure 9-22 lprof Output for the Function CAfind

```
short
CAfind(filedata, searchfunc)
struct caFILEDATA *filedata;
char *searchfunc;
284 [61] {
    short ret_code, findflag;
    unsigned char fname_size;
    char *name;

    284 [66]    CArewind(filedata); /* rewind file pointers */
    284 [67]    findflag = 1;

    404754 [69]    while (findflag)
    40470 [70]        if ((fread((char *) &fname_size,
        sizeof(unsigned char),
        1, filedata->cov_data_ptr)) > 0)
    {
        40470 [72]        name = (char *) malloc(fname_size+1);
        40470 [73]        fread(name, (int)fname_size, 1,
            filedata->cov_data_ptr);
        /* make null-terminated */
        40470 [75]        name[fname_size] = '\0';
        40470 [76]        if (strcmp(name, searchfunc) == 0)
        { /* this is the function, move
            ptr back to beginning of
            function name */
            284 [80]            fseek(filedata->cov_data_ptr,
                -(long)
                (fname_size*sizeof(unsigned char)), 1);
            284 [82]            ret_code = OK;
            284 [83]            findflag = 0;
        }
        else /* this is not it,
            move to next function */
        {
            40186 [87]            if (fname_size != EOD)
            40186 [88]                if (CAjump(filedata->cov_data_ptr)
                == EOF_FAIL)
            { /* error - end of file found */
                0 [90]                ret_code = FUNC_FAIL;
                0 [91]                findflag = 0;
            }
        }
        40470 [94]        free(name);
    }
    else
    { /* end of file before function found */
        0 [98]        ret_code = FUNC_FAIL;
        0 [99]        findflag = 0;
    }
    284 [101]    return(ret_code);
    0 [102] }
```

CAfind searches the data file for data pertaining to a particular function. A data file consists of a header section, followed by a section for each function and an end-of-data marker at the end of the file. The coverage data (execution count) for each function is recorded alongside the function's name.

Notice that the **while** loop (shown between lines 70 and 94) was executed 40,470 times; for 284 successful searches, there were 40,186 unsuccessful searches. We were getting a low rate of return for computing resources spent. A look at the **while** loop also shows why **fread** was executed so many times: the loop contains two calls to **fread**. (See lines 70 and 73 of the **lprof** output.) Finally, the **prof** output reports that **CAjump** was called 40,186 times, once for each unsuccessful search.

Our goals were to minimize the number of unsuccessful searches and, if possible, to decrease the number of calls to **fread**, because these are relatively expensive.

The **lprof** algorithm for merging files consists of two steps: traversing the functions in one of the files sequentially, and calling **CAfind** to locate the data for a given function in the other coverage file.

The first thing that happens in **CAfind** is the resetting of the file pointers so that they point to the first function in the file (line 66). Then, because the given function (which was passed to **CAfind** as an argument) has not been found, the next function in the file is examined to see if it is the correct function. If it is, we are finished. If not, we can skip over the data and try the next function. If we have reached the end of the file, there will be no data for that function in the coverage file and we will return with a failure. By itself, **CAfind** looked fine and there did not seem to be much we could do to improve its performance.

However, by understanding the entire program, we were able to observe that, in almost all situations, the order of the coverage data in the two files to be merged was identical. This meant that on subsequent calls to **CAfind**, the next function being sought was immediately after the one found on the last call to **CAfind**. The original implementation did not take advantage of the fact that the search was usually sequential. The file pointers were always reset to the beginning of the file before the search began. Because the functions were in sequential order, this meant that each successive search took progressively longer.

Profiling Examples

We changed the search strategy so that instead of starting at the beginning of the file on each call to **CAfind**, we started at the place in the file where the previous search had ended. This could have been anywhere in the file. Because files being merged are usually identical, the function being sought is almost always the function following the last one found.

The new search strategy required a slightly more complicated algorithm. Whereas the original strategy demanded only that we check for the end of the file, the new strategy required that we both check for the end of the file and keep track of our current location. The need to do both arose from the sequence of events involved in this type of searching.

The new strategy dictated that each iteration of searching begin where the last search ended. **CAfind** was to search until the function being sought was found. If **CAfind** reached the end of the file before finding that function, it had to continue the search between the first line of the file and the place where it had started the search. Thus **CAfind** had to keep track of when the end of the file was reached. Because the goal of the new strategy was to start each search iteration at the place where the last search had ended, it was obviously necessary to keep track of our current location in the file.

The following screen shows the code for **CAfind** after we changed it to accommodate our new strategy.

Figure 9-23 lprof Output for New Version of Function CAfind

```

short
CAfind(filedata, searchfunc)
struct caFILEDATA *filedata;
char *searchfunc;
284 [61] {
    short ret_code;
    unsigned char fname_size;
    char *name;
    long init_loc;

284 [67]     init_loc = -1;
284 [68]     while (C) {
284 [69]         if (init_loc == -1) {
                /* first time through */
284 [71]             init_loc = ftell(filedata->cov_data_ptr);
            }
            else {
                /* have we wrapped completely around? */
0 [75]             if (ftell(filedata->cov_data_ptr)
                == init_loc) {
                    /* searched all functions */
0 [77]                     ret_code = FUNC_FAIL;
0 [78]                     break;
                }
            }

284 [81]         if ((fread((char *) &fname_size,
                sizeof(unsigned char),
                1, filedata->cov_data_ptr)) > 0) {
284 [83]             if (fname_size == EOD) {
                    /* wrap around to beginning */
0 [85]                     CArewind(filedata);
                    /* go back to top of loop */
                    continue;
            }

284 [89]             name = (char *) malloc(fname_size+1);

```

Profiling Examples

continued

```
284 [90]      fread(name, (int)fname_size, 1,
              filedata->cov_data_ptr);
              /* make null-terminated */
284 [92]      name[fname_size] = '\0';
284 [93]      if (strcmp(name, searchfunc) == 0)
              {
                  /* this is the function, move
                     ptr back to beginning of
                     function name */
284 [97]      fseek(filedata->cov_data_ptr,
                     -(long)
                     (fname_size+
                     sizeof(unsigned char)), 1);
284 [99]      ret_code = OK;
                     break;
              }
              else /* this is not it, move to next
                     function */
              {
                  if (CAjump(filedata->cov_data_ptr)
                      == EOF_FAIL)
                  {
                      /* error - end of file found */
284 [106]      ret_code = FUNC_FAIL;
                      break;
                  }
              }
284 [110]      free(name);
              }
              else
              { /* end of file before function found */
284 [114]      ret_code = FUNC_FAIL;
                     break;
              }
              }

284 [119]      return(ret_code);
0 [120]      }
```

Note that not only did we greatly reduce the number of calls to **fread**, but in typical situations we eliminated calls to **CAjump** entirely. Remember, **CAjump** originally took 3.69 seconds (9.5% of the total execution time), which was more than any other user function.

The **prof** output for the new version is shown in the following screen.

Figure 9-24 **prof** Output for New Version of **lprof**

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
25.4	0.54	0.54	298	1.81	read
11.7	0.25	0.79	2002	0.125	malloc
10.6	0.22	1.01	2848	0.079	fread
8.9	0.19	1.20	579	0.33	lseek
7.0	0.15	1.35	1518	0.099	fwrite
6.1	0.13	1.48			_mcount
4.2	0.09	1.57	569	0.16	CAread
3.8	0.08	1.65	4369	0.018	memcpy
2.8	0.06	1.71	284	0.21	CAor
2.8	0.06	1.77	2	30.	creat
2.8	0.06	1.83	1	60.	CAcov_join
1.9	0.04	1.87	284	0.14	CAfind
1.9	0.04	1.91	284	0.14	CAdata_entry
1.9	0.04	1.95	1717	0.023	free
1.4	0.03	1.98	7	4.	open

The execution time for **CAfind** decreased from 2.99 seconds to 0.04 seconds, and for **CAjump** from 3.69 seconds to 0 seconds. The overall performance for the entire program decreased from forty cpu seconds (two minutes of real time) to two cpu seconds (six seconds of real time).

Improving Test Coverage with **lprof**

It is difficult to write test suites that fully exercise (cover) programs if you have no way of determining how much of the code is exercised. The **lprof** tool removes the guesswork by showing, on a line-by-line basis, which lines of code are executed. This allows the tester to know exactly what has been tested. It also makes it easier to refine and improve tests.

Suppose we want to measure how well a given test suite tests a program. First, we compile the program with **-ql**, so that profiling information will be saved. Then we run the program with the tests to get the profiling data. By looking at the summary output, we can see how much of the code is exercised.

Profiling Examples

Figure 9-25 lprof Summary Output for a Test Suite

Coverage Data Source: test.cnt
Date of Coverage Data Source: Wed Mar 5 11:11:58 1986
Object: myprog

percent covered	lines covered	total lines	function name
91.5	97	106	compile
100.0	18	18	step
100.0	73	73	advance
100.0	4	4	getrnge
42.9	12	28	main
100.0	29	29	execute
100.0	19	19	succeed
42.9	3	7	putdata
0.0	0	19	regerr
100.0	21	21	fgetl
85.2	276	324	TOTAL

More specifically, we can examine individual functions that do not have 100% coverage, to find ways of improving the tests.

The rest of this section consists of three examples that show why certain functions may not have 100% coverage. The first example demonstrates how to uncover an option that is usually missed because it is not documented. Another example shows how to uncover a function that is never called. The third example looks at code that is never executed because of an error condition that is difficult to produce. Each section also explains how to resolve the problem of lack of coverage.

Example 1: Searching for Undocumented Options

First, examine the function **main** to see what parts of the code are not executed.

Figure 9-26 Fragment of Output from **lprof -x**

```

while((c=getopt(argc, argv, "blcnsvi")) != EOF)
  [32]      switch(c) {
            case 'v':
[U] [34]      vflag++;
            break;
            case 'c':
[37]      cflag++;
            break;
            case 'n':
[40]      nflag++;
            break;
            case 'b':
[43]      bflag++;
            break;
            case 's':
[46]      sflag++;
            break;
            case 'l':
[49]      lflag++;
            break;
            case 'i':
[52]      iflag++;
            break;
            case '?':
[U] [55]      errflg++;
[56]      }

```

The output shows that the **-v** option was not tested. By checking the documentation, you can confirm that **-v** is an undocumented option. To correct this, create a test that exercises the **-v** option and add the **-v** option to the manual page.

Example 2: Functions That Are Never Called

None of the lines in the function **regerr** are executed. To find out why, invoke **cscope** and request a list of the functions that call it. The **cscope** tool reports that no function calls **regerr**. Because **regerr** will never be exercised, delete it from the code.

Profiling Examples

Example 3: Hard to Produce Error Conditions

Look at the function `putdata`:

Figure 9-27 Output from `lprof -x` for Function `putdata`

```
void
putdata(output, data)
char    *data;
FILE    *output;

[9]      {
          /* check for file system out of space */
[11]      if (fprintf(output, "%s", data) < 0) {
[U] [12]      fprintf(stderr, "write error with file
          '%s'", filename);
[U] [13]      fclose(output);
[U] [14]      unlink(newreffile);
[U] [15]      exit(C);
          }
[17]      }
```

Because this error is hard to reproduce, it usually does not get tested. However, you can simulate this error by writing your own `fprintf` function that returns a value less than 0. This will cause the error recovery part of the function to get exercised, allowing you to see the following error message:

```
write error with file '@%#&HP'
```

Further inspection reveals that the variable *filename* was never initialized. This oversight caused the error message to be garbled.

Chapter 10

Extended Terminal Interface

Overview 10-1

- How this Chapter is Organized 10-1
- Conventions Used in this Chapter 10-3

What is ETI? 10-5

- The ETI Libraries 10-5
- The ETI/**terminfo** Connection 10-6

Basic ETI Programming 10-8

- What Every ETI Program Needs 10-8
- Compiling an ETI Program 10-11
- Running an ETI Program 10-12
- More about **initscr()** and Lines and Columns 10-13
- More about **refresh()** and Windows 10-13

Simple Input and Output 10-17

- Output 10-17
- Input 10-28
- Output Attributes 10-35
 - How the Color Feature Works 10-39
 - Using the **COLOR_PAIR(*n*)** Attribute 10-41
 - Portability Guidelines 10-43
 - Other Macros and Routines 10-43
 - start_color()** 10-45
 - init_pair()** 10-46
 - init_color** 10-47
- Bells, Whistles, and Flashing Lights: **beep()** and **flash()** 10-48
- Input Options 10-49

Windows 10-53

- Output and Input 10-53
- The Routines **wnoutrefresh()** and **doupdate()** 10-54
- New Windows 10-59
- ETI Low-Level Interface (**curses**) to High-Level Functions 10-61

Panels 10-64

Compiling and Linking Panel Programs 10-65

Creating Panels 10-66

Elementary Panel Window Operations 10-67
 Fetching Pointers to Panel Windows 10-67
 Changing Panel Windows 10-68
 Moving Panel Windows on the Screen 10-69

Moving Panels to the Top or Bottom of the Deck 10-70

Updating Panels on the Screen 10-71

Making Panels Invisible 10-73
 Hiding Panels 10-73
 Reinstating Panels 10-75

Fetching Panels Above or Below Given Panels 10-76

Setting and Fetching the Panel User Pointer 10-78

Deleting Panels 10-81

Menus 10-82

Compiling and Linking Menu Programs 10-83

Overview: Writing Menu Programs in ETI 10-84
 Some Important Menu Terminology 10-84
 What a Menu Application Program Does 10-85
 A Sample Menu Program 10-85

Creating and Freeing Menu Items 10-88

Two Kinds of Menus: Single- and Multi-Valued 10-90
 Manipulating an Item's Select Value in a Multi-Valued Menu 10-90

Manipulating Item Attributes 10-92
 Fetching Item Names and Descriptions 10-92
 Setting Item Options 10-92
 Checking an Item's Visibility 10-94
 Changing the Current Default Values for Item Attributes 10-95

Setting the Item User Pointer 10-96

Creating and Freeing Menus 10-99

Manipulating Menu Attributes 10-101
 Fetching and Changing Menu Items 10-101
 Counting the Number of Menu Items 10-102
 Changing the Current Default Values for Menu Attributes 10-103

Displaying Menus 10-104
 Determining the Dimensions of Menus 10-104
 Associating Windows and Subwindows with Menus 10-112
 Fetching and Changing A Menu's Display Attributes 10-115
 Posting and Unposting Menus 10-117

Menu Driver Processing 10-120
 Defining the Key Virtualization Correspondence 10-120
 ETI Menu Requests 10-122
 Application-Defined Commands 10-126
 Calling the Menu Driver 10-126
 Establishing Item and Menu Initialization and Termination

Routines 10-132
 Fetching and Changing the Current Item 10-135
 Fetching and Changing the Top Row 10-137
 Positioning the Menu Cursor 10-138
 Changing and Fetching the Pattern Buffer 10-139

Manipulating the Menu User Pointer 10-141

Setting and Fetching Menu Options 10-143

Forms 10-146

Compiling and Linking Form Programs 10-147

Overview: Writing Form Programs in ETI 10-148
 Some Important Form Terminology 10-148
 What a Typical Form Application Program Does 10-149
 A Sample Form Application Program 10-149

Creating and Freeing Fields 10-154

Manipulating Field Attributes 10-158
 Obtaining Field Size and Location Information 10-158
 Moving a Field 10-159
 Changing the Current Default Values for Field Attributes 10-160
 Setting the Field Type To Ensure Validation 10-160
 Justifying Data in a Field 10-167

Setting the Field Foreground, Background, and Pad Character 10-169

Some Helpful Features of Fields 10-171

Setting and Reading Field Buffers 10-171

Setting and Reading the Field Status 10-172

Setting and Fetching the Field User Pointer 10-174

Manipulating Field Options 10-177

Creating and Freeing Forms 10-181

Manipulating Form Attributes 10-184

Changing and Fetching the Fields on an Existing Form 10-184

Counting the Number of Fields 10-186

Changing ETI Form Default Attributes 10-186

Displaying Forms 10-187

Determining the Dimensions of Forms 10-187

Associating Windows and Subwindows with a Form 10-189

Posting and Unposting Forms 10-192

Form Driver Processing 10-195

Defining the Virtual Key Mapping 10-196

ETI Form Requests 10-198

Application-Defined Commands 10-204

Calling the Form Driver 10-204

Establishing Field and Form Initialization and Termination

Routines 10-211

Manipulating the Current Field 10-216

Changing the Form Page 10-217

Positioning the Form Cursor 10-219

Setting and Fetching the Form User Pointer 10-221

Setting and Fetching Form Options 10-223

Creating and Manipulating Programmer-Defined Field Types 10-226

Building a Field Type from Two Other Field Types 10-226

Creating a Field Type with Validation Functions 10-227

Freeing Programmer-Defined Field Types 10-230

Supporting Programmer-Defined Field Types 10-231

Other ETI Routines 10-239

Routines for Drawing Lines and Other Graphics 10-240

Routines for Using Soft Labels 10-242

Working with More than One Terminal 10-244

Working with **terminfo** Routines 10-246

What Every **terminfo** Program Needs 10-246

Compiling and Running a **terminfo** Program 10-247

An Example **terminfo** Program 10-248

Working with the **terminfo** Database 10-252

Writing Terminal Descriptions 10-252

Basic Capabilities 10-257

Screen-Oriented Capabilities 10-257

Keyboard-Entered Capabilities 10-258

Parameter String Capabilities 10-259

Comparing or Printing **terminfo** Descriptions 10-262

Converting a **termcap** Description to a **terminfo** Description 10-262

TAM Transition Library 10-264

Compiling and Running TAM Applications under ETI 10-265

Tips for Polishing TAM Application Programs Running under ETI 10-266

How the TAM Transition Library Works 10-267

Translations from TAM Calls to ETI Calls 10-267

The TAM Transition Keyboard Subsystem 10-270

Program Examples 10-274

The **editor** Program 10-274

The **highlight** Program 10-280

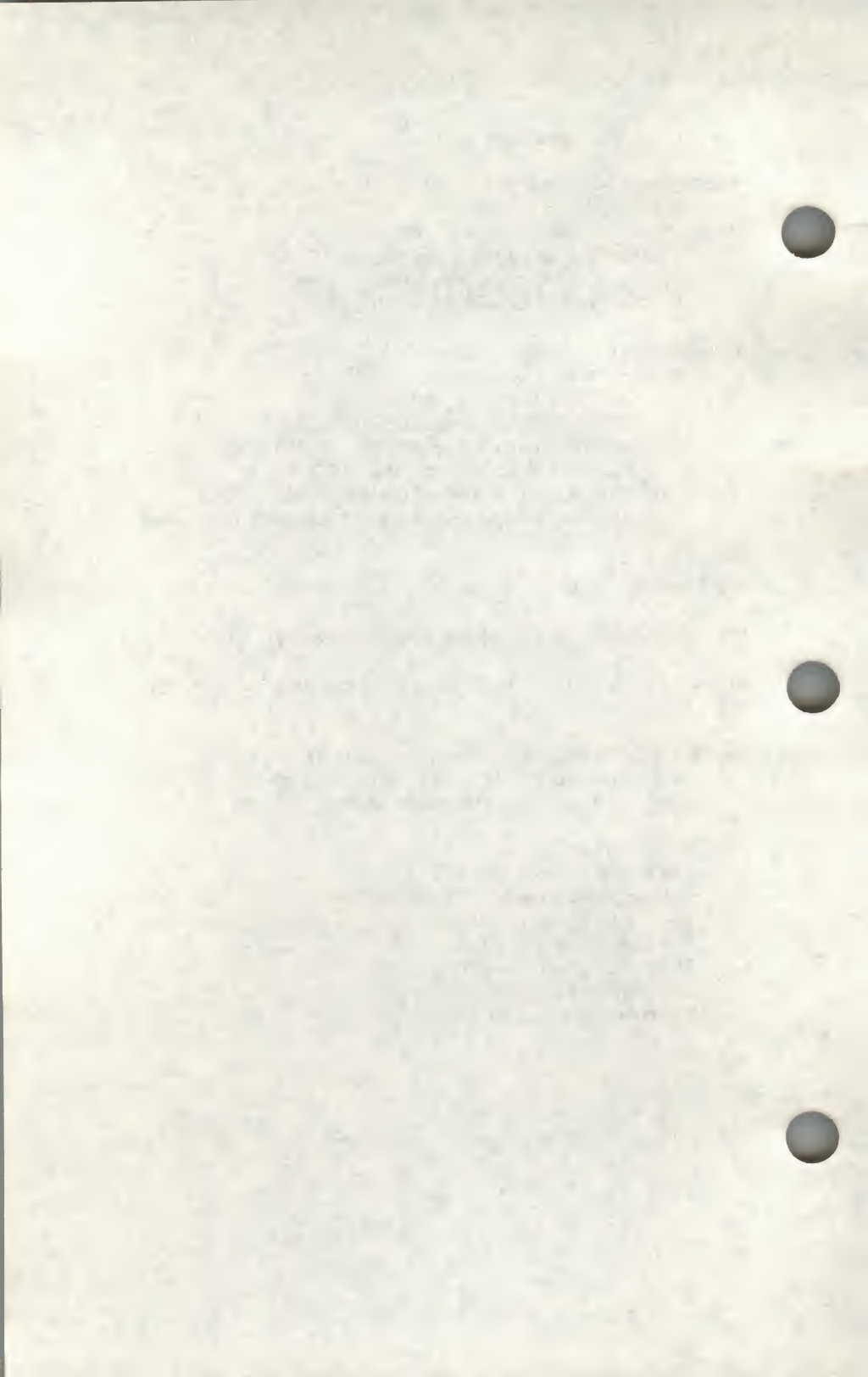
The **scatter** Program 10-282

The **show** Program 10-284

The **two** Program 10-286

The **window** Program 10-288

The **colors** Program 10-290



Overview

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a display, divide a terminal screen into windows, or change the definition of colors. Many screen management programs build end-user terminal interfaces to help users enter and retrieve information from a database — interfaces such as forms, menus, and help and error message displays.

This chapter explains how to use the Extended Terminal Interface (ETI) package to write screen management programs on a UNIX System V/386 system. (It also tells you what you need to know about the **terminfo** database to use ETI.) To start you writing screen management programs as soon as possible, the information in this chapter does not cover every routine in the libraries. Although it covers all routines in the high-level libraries (those that build panels, menus, and forms), it covers only the most frequently used routines in the low-level library (**curses**). For more information, this chapter points you to the **curses(S)**, **terminfo(F)**, and other manual pages in the *Programmer's Reference*. Keep this manual close at hand; you'll find it invaluable when you want to know more about these and other routines.

Because the routines are compiled C functions, you should be familiar with the C programming language before using ETI. You should also be familiar with the UNIX System/C language standard I/O package (see the *C Library Guide* and the **stdio(S)** manual page of the *Programmer's Reference*). With that knowledge and an appreciation for the UNIX System philosophy of building on the work of others, you can design screen management programs for many purposes.

How this Chapter is Organized

This chapter contains eleven sections:

- Introduction to ETI

This is the present section. It briefly describes the ETI libraries and how ETI works with the **terminfo** database.

Overview

- Basic ETI Programming

This section describes the routines and other components that every ETI program needs to work properly, tells you how to compile and run low-level ETI (**curses**) programs, and introduces important concepts such as refreshing.

- Simple Input and Output

This section describes the routines in the low-level ETI (**curses**) library for writing to, and reading from, a screen and manipulating colors. It also covers the suite of video attributes and options which enable you to enhance your displays with striking visual effects.

- Windows

This section explains the use of windows and subwindows. It delves more deeply into the refresh operation and covers the functions **wnoutrefresh()** and **doupdate()**.

- Panels

This section begins the treatment of the high-level ETI functions. It describes the use of panels—windows with interrelationships of depth—and covers the set of panel functions, which enable you to create panels, move them, associate them with different windows, place them on top of other panels, and so forth.

- Menus

This section explains the suite of menu functions. It explains how to create menu items and menus, display them, change menu video attributes, have users interact with menus, and more.

- Forms

This section covers the wealth of form functions. It shows how to create fields and forms, display them, change form video attributes, have users interact with forms, and more.

- Other ETI Routines

This section covers routines for screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time.

- Working with **terminfo** Routines

This section describes a subset of routines in the **curses** library. These routines access and manipulate data in the **terminfo** database. They are used to set up and handle special terminal capabilities such as programmable function keys. This section also describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- Terminal Access Method (TAM) Transition Library

This section explains how to use the TAM transition library and how to rewrite TAM application programs to run efficiently under ETI without the TAM transition library.

- Program Examples

This section includes programs that illustrate uses of low level ETI **curses** routines.

Conventions Used in this Chapter

This section uses the following conventions to discuss ETI routines:

- In program text, the major ETI data types appear in uppercase. They are as follows:
 - **WINDOW**—a rectangular area of the screen treated as a unit
 - **PANEL**—a window with relations of depth to other windows so that regions hidden behind other windows are invisible
 - **ITEM**—a character string consisting of a name and an optional description
 - **MENU**—a screen display that presents a set of items from which the user chooses one or more, depending on the type of menu
 - **FIELD**—an $m \times n$ block of character positions within a form that ETI functions can manipulate as a unit

Overview

- **FORM**—a collection of one or more pages of fields
- **FIELDTYPE**a field attribute that determines what kind of data may occupy the field
- Every ETI function is introduced with a **SYNOPSIS** that describes the type of its arguments and return value, if any. The first line of the **SYNOPSIS** proper describes the routine, while the following lines describe its arguments. On each line, the type of the return value or arguments precedes their names. As an example, consider

SYNOPSIS

```
int set_menu_win (menu, window)
MENU * menu;
WINDOW * window;
```

This says that the function **set_menu_win()** returns a value of type **int** and that it takes two arguments, **menu** and **window**. The argument **menu** is of type **MENU *** (pointer to a menu), while the argument **window** is of type **WINDOW *** (pointer to a window).

- The terms *window*, *panel*, *menu*, and *form* are often shorthand for the phrases *window pointer*, *panel pointer*, *menu pointer*, and *form pointer*, respectively. All ETI routines pass or return pointers to these objects, not the objects themselves.

What is ETI?

ETI is a set of C library routines that promote the development of application programs that display and manipulate windows, panels, menus, and forms and run under the UNIX System. The rest of this section explains the nature of these libraries and the connection between ETI and the **terminfo** library and database.

The ETI Libraries

ETI consists of the following libraries.

- low-level (**curses**)
- **panel**
- **menu**
- **form**
- TAM Transition.

The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw()** that behaves much like **printf(S)** and another routine **getch()** that behaves like **getc(S)**. The automatic teller program at your bank might use **printw()** to print its menus and **getch()** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX System screen editor **vi(C)** (see the *User's Reference*) might also use these and other ETI routines.

A major feature of ETI is cursor optimization. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with ETI routines and edited the sentence

ETI is a great package for creating forms and menus.

to read

ETI is the best package for creating forms and menus.

What is ETI?

the program would change only the best in place of a great. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which an ETI program is run. This means that ETI can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX System's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple ETI program. It uses some of the basic ETI routines to move a cursor to the middle of a terminal screen and print the character string **BullsEye**. Each of these routines is described later in this section. For now, just look at their names, and you will get an idea of what each of them does:

Figure 10-1 A Simple ETI Program

```
#include < curses.h>

main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

The ETI/terminfo Connection

terminfo is both a set of routines that make use of the capabilities of a wide range of terminals and a database that contains descriptions of the terminals that can be used with ETI. Its use as a database is our concern here. See the section "Working with **terminfo** Routines" for details on its use as a set of routines.

A screen management program with ETI routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

Suppose, for instance, that you are using an AT&T Teletype 5425 terminal to run the simple ETI program shown in Figure 10-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the **BullsEye** in the middle of it. The description of the AT&T Teletype 5425 in the **terminfo** database has this information, as well as other information about the terminal's capabilities and how it performs various operations — for example, how its control characters are interpreted. All ETI needs to know before it goes looking for the information is the name of your terminal.

You tell the program the name by putting it in the environment variable **\$TERM** when you log in or by setting and exporting **\$TERM** in your **.profile** file [see **profile(F)**]. Knowing **\$TERM**, an ETI program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following lines are in a **.profile**:

```
TERM=5425
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile(F)** in the *Programmer's Reference*.) The third line of the example tells the UNIX System to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **\$TERM** must be defined and exported first, so that when **tput(C)** [see the *User's Reference*] is called the proper initialization for the current terminal will take place.) If you had these lines in your **.profile** and you ran an ETI program, the program would get the information that it needs about your terminal from the file **/usr/lib/terminfo/5/5425** in the database, which provides a match for **\$TERM**. For more information about the **terminfo** database, see the section "Working with **terminfo** Routines."

Basic ETI Programming

This section describes the low-level routines and other components that every ETI program needs to work properly. It tells you how to compile and run ETI applications using the low-level libraries and introduces important concepts (such as refreshing) that recur throughout this document.

What Every ETI Program Needs

All ETI programs need to include the header files `<menu.h>`, `<form.h>`, and `<panel.h>` and call the routines `initscr()`, `refresh()` or similar routines, and `endwin()`. Some of the other header files, however, include file `curses.h`.

The Header File `<curses.h>`

The header files `<menu.h>`, `<form.h>`, and `<panel.h>` define several global variables and data structures.

To begin, let's consider the variables and data structures defined. `<curses.h>`, among other things, defines the integer variables `LINES` and `COLS`; when an ETI program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine `initscr()` described below.

The integer variables `COLORS` and `COLOR_PAIRS` are also defined in `<curses.h>`. These will be assigned, respectively, the maximum number of colors and color-pairs the terminal can support. These variables are initialized by the `start_color()` routine. (See the section "Color Manipulation.")

Note

LINES and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **\$LINES** and **\$COLUMNS**, may be set in a user's shell environment; an ETI program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this section, we will use the **\$** to distinguish them from the C declarations in the **<urses.h>** header file.

For more information about these variables, see the following sections: "The Routines **initscr()**, **refresh()**, and **endwin()**" and "More about **initscr()** and Lines and Columns."

The header files define the integer constants **OK**, **E_OK**, **ERR**, and others listed in the following sections. ETI routines that return **int** values return these constants under the following conditions:

OK	returned if a low-level or panel function completes properly
E_OK	returned if a menu or form function does so
ERR	returned if a low-level or panel function encounters an error

The other error values returned by the high-level functions are described in the appropriate sections below.

Now let's consider the macro definitions. **<urses.h>** defines many ETI routines as macros that call other macros or ETI routines. For instance, the simple routine **refresh()** is a macro. The line

```
#define refresh() wrefresh(stdscr)
```

shows that when **refresh** is called, it is expanded to call the ETI routine **wrefresh()**. In turn, **wrefresh()** (although it is not a macro) calls the two ETI routines **wnoutrefresh()** and **doupdate()**. Many other routines also group two or three routines together to achieve a particular result.

Note

Macro expansion in ETI programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about `<urses.h>`: it automatically includes `<stdio.h>` and the `<termio.h>` tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines `initscr()`, `refresh()`, `endwin()`

The routines `initscr()`, `refresh()`, and `endwin()` initialize a terminal screen to an “in ETI state,” update the contents of the screen, and restore the terminal to an “out of ETI state,” respectively. Consider the simple program introduced earlier and reproduced in Figure 10-2.

Figure 10-2 The Purposes of `initscr()`, `refresh()`, and `endwin()` in a Program

```
#include <urses.h>

main() {
    initscr();      /* initialize terminal settings and <urses.h>
                   data structures and variables */

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();      /* send output to (update) terminal screen */
    addstr("Eye");
    refresh();      /* send more output to terminal screen */
    endwin();       /* restore all terminal settings */ }
```

An ETI program usually starts by calling `initscr()`; your program should call `initscr()` only once. This routine uses the environment variable `$TERM` to determine what terminal is being used. (See the section, “The ETI/`terminfo` Connection,” for details.) It then initializes all the declared data structures and other variables from `<urses.h>`. For example, `initscr()` would initialize `LINES` and `COLS` for the sample program on whatever terminal it was run. If the TELETYPE 5425 terminal were used, this routine would initialize `LINES` to 24 and `COLS` to 80. Finally, this routine writes error messages to `stderr` and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. The line

```
addstr("Bulls");
```

says to write the character string `Bulls`. For example, if the TELETYPE 5425 terminal were used, these routines would position the cursor and write the character string at (11,36).

Note

All ETI routines that move the cursor move it from its home position in the upper left corner of a screen. The (LINES,COLS) coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the common 'x,y' order of screen (or graph) coordinates. The `-1` in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like **move()** and **addstr()** do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh()** is called after one or more windows (internal representations of the screen) are updated. This is a very important concept, which we discuss below under "More about **refresh()** and Windows."

Finally, an ETI program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling an ETI Program

You compile programs that include ETI routines as C language programs. This means that you use the **cc(CP)** command (documented in the *Programmer's Reference*) to invoke the C compiler.

Basic ETI Programming

The routines are usually stored in the library `/usr/lib/libX.a`, where *X* signifies either **curses**, **panel**, **menu**, or **form**, depending on which library your program needs. To direct the link editor to search this library, you must use the `-l` option with the `cc` command.

The general command line for compiling an ETI program follows:

```
cc file.c [-lX] -lcurses -o file
```

where *X* is either **panel**, **menu**, or **form**; *file.c* is the name of the source program; and *file* is the executable object module. See the appropriate section below for more information.

Using the TAM Transition Library

Some users may have applications using the TAM library routines that originally ran on the UNIX System PC. "The TAM Transition Library," Appendix B of this document, explains how to compile and run these applications.

Running an ETI Program

ETI programs count on certain information being in a user's environment to run properly. Specifically, users of a low-level ETI program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, turn again to the section "The ETI/**terminfo** Connection" in this chapter. Users of an ETI program could also define the environment variables **\$LINES**, **\$COLUMNS**, and **\$TERMINFO** in their **.profile** files. However, unlike **\$TERM**, these variables do not have to be defined.

If an ETI program does not run as expected, you might want to debug it with **sdb**(CP), which is documented in the *Programmer's Reference*. When using **sdb**, you have to keep a few points in mind. First, an ETI program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the ETI program is not aware.

Second, an ETI program doesn't output to a window until **refresh()** or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on ETI routines that are macros, such as **refresh()**, does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh()** instead of **refresh()**. See the above section, "The Header File **< curses.h >**," for more information about macros.

More about **initscr()** and Lines and Columns

After determining a terminal's screen dimensions, **initscr()** sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **\$LINES** and **\$COLUMNS**.

More about **refresh()** and Windows

As mentioned above, ETI routines do not update a terminal until **refresh()** is called. Instead, they write to an internal representation of the screen called a window. When **refresh()** is called, all the accumulated output is sent from the window to the current terminal screen.

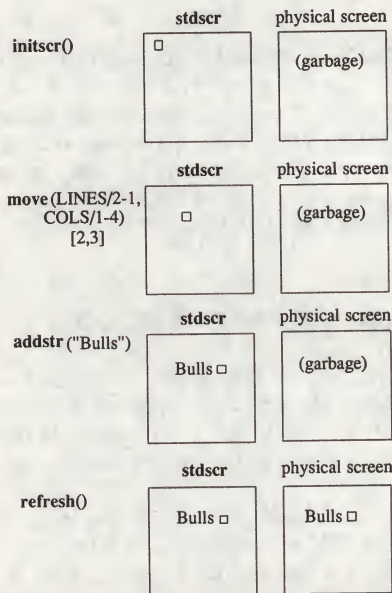
A window acts a lot like a buffer does when you use a UNIX System editor. When you invoke **vi(C)** (see the *User's Reference*), for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of ETI routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh()** is called.

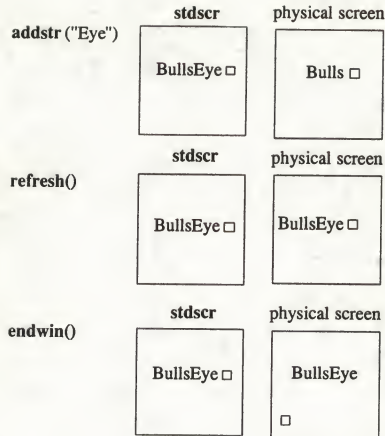
< curses.h > supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using ETI routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh()** is called, it compares the two screen images and sends a stream of characters to the terminal that make the physical screen look like **stdscr**. An ETI program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on

Basic ETI Programming

the window (**stdscr**). It optimizes output by printing as few characters as possible. Figure 10-3 illustrates what happens when you execute the sample ETI program that prints BullsEye at the center of a terminal screen. Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh()** is called. This **refresh()** clears the screen and updates it with the current contents of **stdscr**.

Figure 10-3 The Relationship between **stdscr** and a Terminal Screen





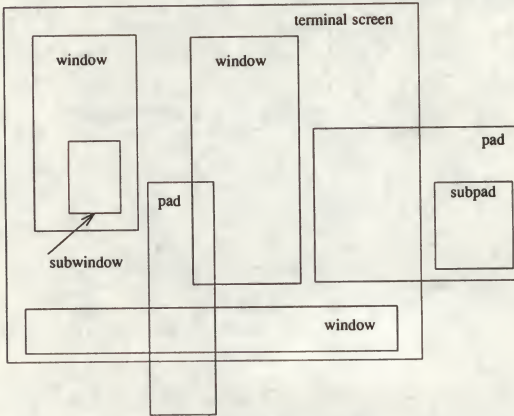
You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window. It's possible to subdivide a screen into many windows, refreshing each one of them as desired. And it's possible to create a window within a window; the smaller window is called a subwindow. See the section, "Windows," for more information.

Pads

Some ETI routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 10-4 represents what a pad, a subwindow, and some other windows could look like in comparison to a physical screen.

Figure 10-4 Multiple Windows and Pads Mapped to a Physical Screen



The later section “Windows” describes the routines you use to create and use windows and pads. If you’d like to see an ETI program with windows now, turn to the **window** program under the section “ETI Program Examples” in this chapter.

Simple Input and Output

This section explains the numerous functions that enable you to do I/O under the ETI environment. It also covers the set of video attributes and options which can enhance ETI output with striking visual effects.

Output

The routines that low-level ETI provides for writing to **stdscr** are similar to those provided by the **stdio(S)** library for writing to a file. They let you

- write a character at a time — **addch()**
- write a string — **addstr()**
- format a string from a variety of input arguments — **printw()**
- move a cursor or move a cursor and print character(s) — **move()**, **mvaddch()**, **mvaddstr()**, **mvprintw()**
- clear a screen or a part of it — **clear()**, **erase()**, **clrtoeol()**, **clrtoeol()**

Following are descriptions and examples of these routines. The ETI library provides its own set of input and output functions. You should not use other I/O routines or system calls, like **printf(S)** and **scanf(S)**, in an ETI program. They may cause undesirable results when you run the program.

Simple Input and Output

addch()

SYNTAX

```
#include <curses.h>

int addch(ch)
    chtype ch;
```

NOTES

- **addch()** writes a single character to **stdscr** and advances the cursor to the next character position.
- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** contains data and attributes (see “Output Attributes” in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **unsigned long**) that **chtype** is declared to be in **<curses.h>**. This will ensure future compatibility.
- **addch()** does some translations. For example, it converts
 - the **<NL>** character to a clear to end of line and a move to the next line
 - the tab character to an appropriate number of blanks
 - other control characters to their **^X** notation
- **addch()** normally returns **OK**. The only time **addch()** returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.
- **addch()** is a macro.

EXAMPLE

```
#include < curses.h>

main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

The output from this program will appear as follows, with 'a' in position 0, 0:

a

\$□

See also the **show** program under “ETI Example Programs” in this chapter.

Simple Input and Output

addstr()

SYNTAX

```
#include <curses.h>
int addstr(str)
char *str;
```

NOTES

- **addstr()** writes a string of characters to **stdscr**.
- **addstr()** calls **addch()** to write each character.
- **addstr()** follows the same translation rules as **addch()**.
- **addstr()** returns **OK** on success and **ERR** on error.
- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string `BullsEye`. See Figures 10-2, 10-3, and 10-4.

printw()

SYNTAX

```
#include <curses.h>
int printw(fmt [,arg...])
char *fmt
```

NOTES

- **printw()** handles formatted printing on **stdscr**.
- Like **printf**, **printw()** takes a format string and a variable number of arguments.
- Like **addstr()**, **printw()** calls **addch()** to write the string.
- **printw()** returns **OK** on success and **ERR** on error.

Simple Input and Output

EXAMPLE

```
#include <curses.h>

main()
{
    char* title = "Not specified";
    int no = 0;

    /* Missing code. */

    initscr();

    /* Missing code. */

    printw("%s is not in stock.\n", title);
    printw("Please ask the cashier to order %d for you.\n", no);

    refresh();
    endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.
```

\$□

move()

SYNTAX

```
#include < curses.h>
```

```
int move(y, x);
```

```
int y, x;
```

NOTES

- **move()** positions the cursor for **stdscr** at the given row **y** and the given column **x**.
- Notice that **move()** takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section "The Routines **initscr()**, **refresh()**, and **endwin()**" for more information.
- **move()** may be combined with the write functions to form
 - **mvaddch(y, x, ch)**, which moves to a given position and prints a character
 - **mvaddstr(y, x, str)**, which moves to a given position and prints a string of characters
 - **mvprintw(y, x, fmt [,arg...])**, which moves to a given position and prints a formatted string.
- **move()** returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.
- **move()** is a macro.

Simple Input and Output

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here -->□if move() works.
```

```
Press <CR> to end test.
```

After you press <CR>, the screen looks like this:

```
Cursor should be here --> if move() works.
```

```
Press <CR> to end test.
$□
```

See the **scatter** program under “ETI Program Examples” in this chapter for another example using **move()**.

clear() and erase()**SYNTAX**

```
#include <curses.h>

int clear()
int erase()
```

NOTES

- Both routines change **stdscr** to all blanks.
- **clear()** assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the low-level ETI or **curses(S)** manual page for more information about **clearok()**.
- **initscr()** automatically calls **clear()**.
- **clear()** and **erase()** always return OK.
- Both routines are macros.

EXAMPLE for clear()

```
#include <curses.h>

main()
{
    initscr();
    printw("Press <CR> to clear the screen ");
    move(0,30);
    refresh();
    getch();
    clear();
    refresh();
    endwin();
}
```

Simple Input and Output

EXAMPLE for erase()

```
#include <urses.h>

main()
{
    initscr();
    printw("Press <CR> to erase the screen ");
    move(0,30);
    refresh();
    getch();
    erase();
    refresh();
    endwin();
}
```

clrtoeol() and clrtobot()

SYNTAX

```
#include <urses.h>

int clrtoeol()
int clrtobot()
```

NOTES

- **clrtoeol()** changes the remainder of a line to all blanks.
- **clrtobot()** changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

EXAMPLE

The following sample program uses `clrrobot()`.

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrrobot();
    refresh();
    endwin();
}
```

Here's the output generated by running this program:

.....A

```
Press <CR> to delete from here□to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh()`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press `<CR>`:

```
Press <CR> to delete from here
$□
```

See the **show** and **two** programs under "Program Examples" for other uses of `clrtoeol()`.

Input

Low-level routines for reading from the current terminal are similar to those provided by the **stdio(S)** library for reading from a file. They let you do the following:

- read one character at a time — **getch()**
- read a <NL>-terminated string — **getstr()**
- parse input, converting and assigning selected data to an argument list — **scanw()**

The primary routine is **getch()**, which processes a single input character and then returns that character. This routine is like the C library routine **getchar()**(S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar()**. For example, you can use **getch()** with the ETI routine **keypad()**, which allows a low-level ETI program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch()** and **keypad()** on the **curses(S)** manual page for more information about **keypad()**.

The following pages describe and give examples of the basic routines for getting input in a screen program.

getch()**SYNTAX****#include <curses.h>****int getch()****NOTES**

- **getch()** reads a single character from the current terminal.
- **getch()** returns the value of the character or **ERR** on 'end of file,' receipt of signals, or nonblocking read with no input.
- **getch()** is a macro.
- See the discussions about **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**, **noraw()**, **halfdelay()**, **nodelay()**, and **keypad()** on the following pages and in **curses(S)**.

EXAMPLE**#include <curses.h>****main()**

{

int ch; **initscr();** **cbreak();** /* Explained later in the section "Input Options" */ **addstr("Press any character: ");** **refresh();** **ch = getch();** **printw("\n\nThe character entered was a '%c'.\n", ch);** **refresh();** **endwin();**

}

The output from this program follows. The first **refresh()** sends the **addstr()** character string from **stdscr** to the terminal.

Press any character: □

Simple Input and Output

Now assume that a **w** is typed at the keyboard. **getch()** accepts the character and assigns it to **ch**. Finally, the second **refresh()** is called and the screen appears as follows:

```
Press any character:  w
```

```
The character entered was a 'w'.
```

```
$□
```

For another example of **getch()**, see the **show** program under “Program Examples” in this chapter.

getstr()

SYNTAX

```
#include <curses.h>

int getstr(str)
char *str;
```

NOTES

- **getstr()** reads characters and stores them in a buffer until a <CR>, <NL>, or <ENTER> is received from **stdscr**. **getstr()** does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr()** is a macro; it calls **getch()** to read each character.
- **getstr()** returns **ERR** if **getch()** returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions on **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**, **noraw()**, **halfdelay()**, **nodelay()**, and **keypad()** on the following pages and in ETI **curses(S)**.

Simple Input and Output

EXAMPLE

```
#include <courses.h>

main()
{
    char str[256];

    initscr();
    cbreak(); /* Explained later in the section "Input Options" */
    addstr("Enter a character string terminated by <CR>:\n\n");
    refresh();
    getstr(str);
    printw("\n\nThe string entered was '%s'\n", str);
    refresh();
    endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX System.'
The final screen (after entering <CR>) would appear as follows:

Enter a character string terminated by <CR>:

I enjoy learning about the UNIX System.

The string entered was

'I enjoy learning about the UNIX System.'

\$□

scanw()

SYNTAX

```
#include <curses.h>
int scanw(fmt [, arg...])
char *fmt;
```

NOTES

- **scanw()** calls **getstr()** and parses an input line.
- Like **scanf(S)**, **scanw()** uses a format string to convert and assign to a variable number of arguments.
- **scanw()** returns the same values as **scanf()**.
- See **scanf(S)** for more information.

Simple Input and Output

EXAMPLE

```
#include <courses.h>

main()
{
    char string[100];
    float number;

    initscr();
    cbreak();          /* Explained later in the */
    echo();             /* section "Input Options" */
    addstr("Enter a number and a string separated by a comma: ");
    refresh();
    scanw("%f,%s",&number,string);
    clear();
    printw("The string was \"%s\" and the number was %f.",string,number);
    refresh();
    endwin();
}
```

Notice the two calls to **refresh()**. The first call updates the screen with the character string passed to **addstr()**, the second with the string returned from **scanw()**. Also notice the call to **clear()**. Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.
```

```
$□
```

Output Attributes

When we talked about `addch()`, we said that it writes a single character of the type `chtype` to `stdscr`. `chtype` has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, a particular color, underlined, and so on.

`stdscr` always has a set of current attributes that it associates with each character as it is written. However, using the routine `attrset()` and related ETI routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- `A_BLINK`—blinking
- `A_BOLD`—extra bright or bold
- `A_DIM`—half bright
- `A_REVERSE`—reverse video
- `A_STANDOUT`—a terminal's best highlighting mode
- `A_UNDERLINE`—underlining
- `A_ALTCHARSET`—alternate character set (see the section "Drawing Lines and Other Graphics" in this chapter)
- `COLOR_PAIR(n)`—change foreground and background colors (see the section on "Color Manipulation" in this section)

To use these attributes, you must pass them as arguments to `attrset()` and related routines; they can also be ORed with the bitwise OR (`|`) to `addch()`.

Note

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, an ETI program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

Simple Input and Output

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Attributes can be turned on singly, such as `attrset(A_BOLD)` in the example, or in combination. To turn on blinking bold text, for example, you would use `attrset(A_BLINK | A_BOLD)`. Individual attributes can be turned on and off with the ETI routines `attron()` and `attroff()` without affecting other attributes. `attrset(0)` turns all attributes off, including changes you may have made to foreground and background color.

Notice the attribute called `A_STANDOUT`. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the `A_STANDOUT` attribute is recommended. Two convenient functions, `standout()` and `standend()` can be used to turn on and off this attribute. `standend()`, in fact, turns off all attributes.

In addition to the attributes listed above, there are two bit masks called `A_CHARTEXT` and `A_ATTRIBUTES`. You can use these bit masks with the ETI function `inch()` and the C logical AND (`&`) operator to extract the character or attributes of a position on a terminal screen. See the discussion of `inch()` on the `curses(S)` manual page. A third bit mask, `A_COLOR`, can be used to extract information about the color-pair field of a position on a terminal screen.

Following are descriptions of `attrset()` and the other ETI routines that you can use to manipulate attributes.

attron(), **attrset()**, and **attroff()**

SYNTAX

```
#include <curses.h>

int attron( attrs )
  chtype attrs;

int attrset( attrs )
  chtype attrs;

int attroff( attrs )
  chtype attrs;
```

NOTES

- **attron()** turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.
- **attrset()** turns on the requested attributes **attrs** instead of any that are currently turned on.
- **attroff()** turns off the requested attributes **attrs** if they are on.
- The attributes may be combined using the bitwise OR (|).
- All return **OK**.

EXAMPLE

See the **highlight** program under “Program Examples” in this chapter.

Simple Input and Output

standout() and **standend()**

SYNTAX

```
#include <curses.h>

int standout()
int standend()
```

NOTES

- **standout()** turns on the preferred highlighting attribute, **A_STANDOUT**, for the current terminal. This routine is equivalent to **attron(A_STANDOUT)**.
- **standend()** turns off all attributes. This routine is equivalent to **attrset(0)**, where **attrset()** takes the argument **0**.
- Both always return **OK**.

EXAMPLE

Again, see the **highlight** program under “Program Examples” in this chapter.

Color Manipulation

The ETI color manipulation routines allow you to use colors on an alphanumeric terminal as you would use any other video attribute. You can find out if the ETI library on your system supports the color routines by checking the file `/usr/include/curses.h` to see if it defines the macro `COLOR_PAIR(n)`.

This section begins with a description of the color feature at a general level. Then, the use of color as an attribute is explained. Next, the ways to define color-pairs and change the definitions of colors is explained. Finally, there are guidelines for ensuring the portability of your program, and a section describing the color manipulation routines and macros, with examples.

How the Color Feature Works

Colors are always used in pairs, consisting of a foreground color (used for the character) and a background color (used for the field on which the character is displayed). ETI uses this concept of color-pairs to manipulate colors. In order to use color in a ETI program, you must first define (initialize) the individual colors, then create color-pairs using those colors, and finally, use the color-pairs as attributes.

Actually, the process is even simpler, since ETI maintains a table of initialized colors for you. This table has as many entries as the number of colors your terminal can display at one time. Each entry in the table has three fields: one each for the intensity of the red, green, and blue components in that color.

Note

ETI uses RGB (Red, Green, Blue) color notation. This notation allows you to specify directly the intensity of red, green, and blue light to be generated in an additive system. Some terminals use an alternative notation, known as HSL (Hue, Saturation, Luminosity) color notation. Terminals that use HSL can be identified in the **terminfo** database, and ETI will make conversions to RGB notation automatically.

Simple Input and Output

At the beginning of any ETI program that uses color, all entries in the colors table are initialized with eight basic colors, as follows:

	Intensity of Component		
	(R)ed	(G)reen	(B)lue
/* black: 0 */	0	0	0
/* blue: 1 */	0	0	1000
/* green: 2 */	0	1000	0
/* cyan: 3 */	0	1000	1000
/* red: 4 */	1000	0	0
/* magenta: 5 */	1000	0	1000
/* yellow: 6 */	1000	1000	0
/* white: 7 */	1000	1000	1000

The Default Colors Table

Most color alphanumeric terminals can display eight colors at the same time, but if your terminal can display more than eight, then the table will have more than eight entries. The same eight colors will be used to initialize additional entries. If your terminal can display only N colors, where N is less than eight, then only the first N colors shown in the Colors Table will be used.

You can change these color definitions with the routine `init_color()`, if your terminal is capable of redefining colors. If your terminal is not able to change the definition of a color, use of `init_color()` returns `ERR`.

The following color macros are defined in `curses.h` and have numeric values corresponding to their position in the Colors Table.

<code>COLOR_BLACK</code>	0
<code>COLOR_BLUE</code>	1
<code>COLOR_GREEN</code>	2
<code>COLOR_CYAN</code>	3
<code>COLOR_RED</code>	4
<code>COLOR_MAGENTA</code>	5
<code>COLOR_YELLOW</code>	6
<code>COLOR_WHITE</code>	7

ETI also maintains a table of color-pairs, which has space allocated for as many entries as the number of color-pairs that can be displayed on your terminal screen at the same time. Unlike the colors table, however, there are no default entries in the pairs table: it is your responsibility to initialize any color-pair you want to use with `init_pair()`, before you use it as an attribute.

Each entry in the pairs table has two fields: the foreground color, and the background color. For each color-pair that you initialize, these two fields will each contain a number representing a color in the colors table. (Note that color-pairs can only be made from previously initialized colors.)

The following example pairs table shows that a programmer has used `init_pair()` to initialize color-pair 1 as a blue foreground (entry 1 in the default color table) on yellow background (entry 6 in the default color table). Similarly, the programmer has initialized color-pair 2 as a cyan foreground on a magenta background. Not-initialized entries in the pairs table would actually contain zeros, which corresponds to black on black.

Note that color-pair 0 is reserved for use by ETI and should not be changed or used in application programs.

Color-Pair Number	Foreground	Background
0	0	0
1	1	6
2	3	5
3	0	0
4	0	0
5	0	0
.	.	.
.	.	.

Example of a Pairs Table

Two global variables used by the color routines are defined in `<courses.h>`. They are **COLORS**, which contains the maximum number of colors the terminal supports, and **COLOR_PAIRS**, which contains the maximum number of color-pairs the terminal supports. Both are initialized by the `start_color()` routine to values it gets from the `terminfo` database.

Using the `COLOR_PAIR(n)` Attribute

If you choose to use the default color definitions, there are only two things you need to do before you can use the attribute `COLOR_PAIR(n)`. First, you must call the routine `start_color()`. Once you've done that, you can initialize color-pairs with the routine `init_pair(pair, f, b)`. The first argument, *pair*, is the number of the color-pair to be initialized (or changed), and must be between 1 and `COLOR_PAIRS-1`. The arguments *f* and *b* are the foreground color number and the background color number. The value of these arguments must be between 0 and

Simple Input and Output

COLORS-1. For example, the two color-pairs in the pairs table described earlier can be initialized in the following way:

```
init_pair (1, COLOR_BLUE, COLOR_YELLOW);  
init_pair (2, COLOR_CYAN, COLOR_MAGENTA);
```

Once you've initialized a color-pair, the attribute **COLOR_PAIR(*n*)** can be used as you would use any other attribute. **COLOR_PAIR(*n*)** is a macro, defined in **<curses.h>**. The argument, *n*, is the number of a previously initialized color-pair. For example, you can use the routine **attron()** to turn on a color-pair in addition to any other attributes you may currently have turned on:

```
attron (COLOR_PAIR(C));
```

If you had initialized color-pair 1 in the way shown in the example pairs table, then characters displayed after you turned on color-pair 1 with **attron()** would be displayed as blue characters on a yellow background.

You can also combine **COLOR_PAIR(*n*)** with other attributes, for example,

```
attrset (A_BLINK | COLOR_PAIR(C));
```

would turn on blinking and whatever you have initialized color-pair 1 to be. (**attron()** and **attrset()** are described in the "Controlling Input and Output" section of this chapter, and also on the **curses(S)** manual page in the *Programmer's Reference*.) If your terminal is capable of redefining colors, you can change the predefined colors with the routine **init_color(*color*, *r*, *g*, *b*)**. The first argument, *color*, is the numeric value of the color you want to change, and the last three, *r*, *g*, and *b*, are the intensities of the red, green, and blue components, respectively, that the new color will contain. Once you change the definition of a color, all occurrences of that color on your screen change immediately.

So, for example, you could change the definition of color 1 (**COLOR_BLUE** by default), to be light blue, in the following way.

```
init_color (COLOR_BLUE, 0, 700, 1000);
```

Portability Guidelines

Like the rest of ETI the color manipulation routines have been designed to be terminal independent. But it must be remembered that the capability of terminals vary. For example, if you write a program for a terminal that can support 64 color-pairs, that program would not be able to produce the same color effects on a terminal that supports at most 8 color-pairs.

When you are writing a program that may be used on different terminals, you should follow these guidelines:

Use at most seven color-pairs made from at most eight colors.

Programs that follow this guideline will run on most color terminals. Only seven, not eight, color-pairs should be used, even though many terminals support eight color-pairs, because **curses** reserves color-pair 0 for its own use.

Do not use color 0 as a background color.

This is recommended because on some terminals, no matter what color you have defined it to be, color 0 will always be converted to black when used for a background.

Combine color and other video attributes.

Programs that follow this guideline will provide some sort of highlighting, even if the terminal is monochrome. On color terminals, as many of the listed attributes as possible would be used. On monochrome terminals, only the video attributes would be used, and the color attribute would be ignored.

Use the global variables `COLORS` and `COLOR-PAIRS` rather than constants when deciding how many colors or color-pairs your program should use.

Other Macros and Routines

There are two other macros defined in `<curses.h>` that you can use to obtain information from the color-pair field in characters of type `chtype`.

- **`A_COLOR`** is a bit mask to extract color-pair information. It can be used to clear the color-pair field, and to determine if any color-pair is being used.
- **`PAIR_NUMBER(attrs)`** is the reverse of **`COLOR_PAIR(n)`**. It returns the color-pair number associated with the named attribute, *attrs*.

Simple Input and Output

There are two color routines that give you information about the terminal your program is running on. The routine **has_colors()** returns a Boolean value: **TRUE** if the terminal supports colors, **FALSE** otherwise. The routine **can_change_colors()** also returns a Boolean value: **TRUE** if the terminal supports colors *and* can change their definitions, **FALSE** otherwise.

There are two color routines that give you information about the colors and color-pairs that are currently defined on your terminal. The routine **color_content()** gives you a way to find the intensity of the RGB components in an initialized color. It returns **ERR** if the color does not exist or if the terminal cannot change color definitions, **OK** otherwise. The routine **pair_content()** allows you to find out what colors a given color-pair consists of. It returns **ERR** if the color-pair has not been initialized, **OK** otherwise.

These routines are explained in more detail on the **curses(S)** manual page in the *Programmer's Reference*.

The routines **start_color()**, **init_color()**, and **init_pair()** are described on the following pages, with examples of their use. You can also refer to the program **colors** in the section "Program Examples," at the end of this chapter, for an example of using the attribute of color in windows.

start_color()

SYNTAX

```
#include <curses.h>

int start_color()
```

NOTES

- This routine must be called if you want to use colors, and before any other color manipulation routine is called. It is good practice to call it right after **initscr()**.
- It initializes eight default colors (black, blue, green, cyan, red, magenta, yellow, and white), and the global variables **COLORS** and **COLOR_PAIRS**. If the value corresponding to **COLOR_PAIRS** in the **terminfo** database is greater than 64, **COLOR_PAIRS** will be set to 64.
- It restores the terminal's colors to the values they had when the terminal was just turned on.
- It returns **ERR** if the terminal does not support colors, **OK** otherwise.

EXAMPLE

See the example under **init_pair()**.

Simple Input and Output

init_pair()

SYNTAX

```
#include <curses.h>

int init_pair (pair, f, b)
short pair, f, b;
```

NOTES

- **init_pair()** changes the definition of a color-pair.
- Color-pairs must be initialized with **init_pair()** before they can be used as the argument to the attribute macro **COLOR_PAIR(n)**.
- The value of the first argument, *pair*, is the number of a color-pair, and must be between **1** and **COLOR_PAIRS-1**.
- The value of the *f* (foreground) and *b* (background) arguments must be between **0** and **COLORS-1**.
- If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will change to the new definition.
- It returns **OK** if it was able to change the definition of the color-pair, **ERR** otherwise.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr ();
    if (start_color () == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        addstr ("Red on Green");
        refresh();
    }
    endwin();
}
```

Also see the program **colors** in the section “Program Examples.”

init_color**SYNTAX**

```
#include <curses.h>
```

```
int init_color(color, r, g, b)
```

```
short color, r, g, b;
```

NOTES

- **init_color()** changes the definition of a color.
- The first argument, *color*, is the number of the color to be changed. The value of *color* must be between **0** and **COLORS-1**.
- The last three arguments, *r*, *g*, and *b*, are the amounts of red, green, and blue (RGB) components in the new color. The values of these three arguments must be between **0** and **1000**.
- When **init_color()** is used to change the definition of an entry in the colors table, all places where the old color was used on the screen immediately change to the new color.
- It returns **OK** if it was able to change the definition of the color, **ERR** otherwise.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    if (start_color() == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        if (init_color (COLOR_RED, 0, 0, 1000) == OK)
            addstr ("BLUE ON GREEN");
        else
            addstr ("RED ON GREEN");
        refresh ();
    }
    endwin();
}
```


Bells, Whistles, and Flashing Lights: `beep()` and `flash()`

Occasionally, you may want to get a user's attention. Two low-level ETI routines are designed to help you do this—they let you ring the terminal's chimes and flash its screen.

`flash()` flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine `beep()` can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to `beep()` will flash the screen.)

SYNTAX

```
#include <curses.h>
```

```
int flash()
```

```
int beep()
```

NOTES

- `flash()` tries to flash the terminal screen, if possible, and, if not, tries to ring the terminal bell.
- `beep()` tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- `beep` will not work if you redefine `TRUE` to something other than `1`.
- Neither returns any useful value.

Input Options

The UNIX System does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character (typically #) and a line kill character (typically @)
- interprets a CTRL-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates <CR> to <NL>

Because an ETI program maintains total control over the screen, low-level ETI turns off echoing on the UNIX System and does echoing itself. At times, you may not want the UNIX System to process other characters in the standard way in an interactive screen management program. Some ETI routines, **noecho()** and **cbreak()**, for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 10-5 shows some of the major routines for controlling input.

Every low-level ETI program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak()**, **raw()**, **nocbreak()**, or **noraw()** mode. Although the low-level ETI program starts up in **echo()** mode, as Figure 10-5 shows, none of the other modes are guaranteed.

The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The ETI routine **noecho()** is designed for this purpose. However, when **noecho()** turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak()** causes these characters to be uninterpreted.

Simple Input and Output

Figure 10-5 Input Option Settings for ETI Programs

Input Options	Characters	
	Interpreted	Uninterpreted
Normal 'out of ETI state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal ETI 'start up state'	echoing (simulated)	All else undefined.
cbreak() and echo()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and noecho()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and noecho()	break, quit stripping erase, kill EOF	echoing
nocbreak() and echo()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of cbreak())		break, quit stripping

Do not use the combination **nocbreak()** and **echo()**. If you use it in a program and also use **getch()**, the program will go in and out of **cbreak()** mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Figure 10-5, you can use the ETI routines **noraw()**, **halfdelay()**, and **nodelay()** to control input. See the **curses(S)** manual page for discussions of these routines.

The next few pages describe **noecho()**, **cbreak()** and the related routines **echo()** and **nocbreak()** in more detail.

echo() and **noecho()**

SYNTAX

```
#include <curses.h>
```

```
int echo()
```

```
int noecho()
```

NOTES

- **echo()** turns on echoing of characters by ETI as they are read in. This is the initial setting.
- **noecho()** turns off the echoing.
- Neither returns any useful value.
- ETI programs may not run properly if you turn on echoing with **nocbreak()**. See Figure 10-5 and accompanying caution. After you turn echoing off, you can still echo characters with **addch()**.

EXAMPLE

See the **editor** and **show** programs under “Program Examples” in this chapter.

Simple Input and Output

cbreak() and **nocbreak()**

SYNTAX

```
#include < curses.h >
int cbreak()
int nocbreak()
```

NOTES

- **cbreak()** turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak()** returns to normal 'line at a time' processing. This is typically the initial setting.
- Neither returns any useful value.
- ETI programs may not run properly if **cbreak()** is turned on and off within the same program or if the combination **nocbreak()** and **echo()** is used.
- See Figure 10-5 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "Program Examples" in this chapter.

Windows

An earlier section in this chapter, “More about **refresh()** and Windows,” explained what windows and pads are and why you might want to use them. This section describes the ETI routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch('c')** would become **waddch(mywin, 'c')** if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in “Simple Input and Output.”

- **waddch(win, ch)**
- **mvwaddch(win, y, x, ch)**
- **waddstr(win, str)**
- **mvwaddstr(win, y, x, str)**
- **wprintw(win, fmt [, arg...])**
- **mvwprintw(win, y, x, fmt [, arg...])**
- **wmove(win, y, x)**
- **wclear(win)** and **werase(win)**
- **wclrtoeol(win)** and **wclrtoeol(win)**
- **wrefresh(win)**

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a **win** argument. Notice that the routines whose names begin with **mvw** take the **win** argument before the **y, x** coordinates, which is contrary to what the

names imply. See **curses(S)** for more information about these routines or the versions of the input routines **getch**, **getstr()**, and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh()** and **wnoutrefresh()** (see below). In place of these two routines, you have to use **prefresh()** and **pnoutrefresh()** with pads.

The Routines **wnoutrefresh()** and **doupdate()**

If you recall from the earlier discussion about **refresh()**, we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see “What Every ETI Program Needs” and “More about **refresh()** and Windows”).

The **wrefresh()** routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh()** and **doupdate()**. Similarly, **prefresh()** sends the contents of a pad to a screen by calling **pnoutrefresh()** and **doupdate()**.

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens more efficiently than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the processor time used. Figure 10-6 shows a sample program that uses only one **doupdate()**.

Figure 10-6 Using `wnoutrefresh()` and `doupdate()`

```
#include <curses.h>

main()
{
    WINDOW *w1, *w2;

    initscr();
    w1 = newwin(2,6,0,3);
    w2 = newwin(1,4,5,4);
    waddstr(w1, "Bulls");
    wnoutrefresh(w1);
    waddstr(w2, "Eye");
    wnoutrefresh(w2);
    doupdate();
    endwin();
}
```

Notice from the sample that you declare a new window at the beginning of an ETI program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named `w1` and `w2` with the routine `newwin()` according to certain specifications. `newwin()` is discussed in more detail below.

Figure 10-7 illustrates the effect of `wnoutrefresh()` and `doupdate()` on these two windows, the virtual screen, and the physical screen.

Windows

Figure 10-7 The Relationship Between a Window and a Terminal Screen

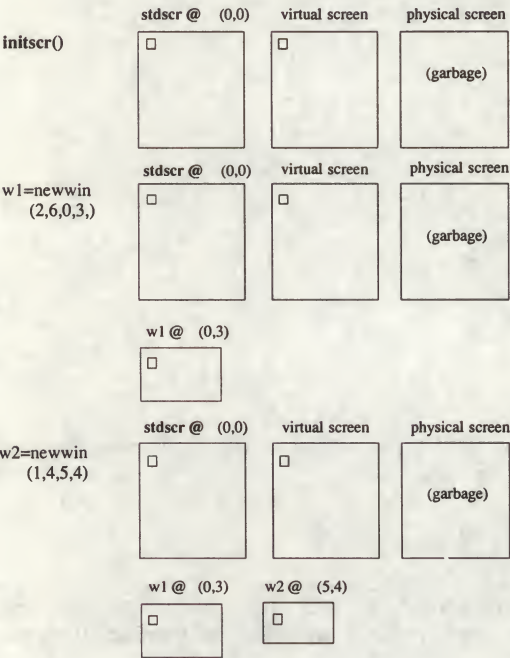


Figure 10-7 The Relationship Between a Window and a Terminal Screen
(continued)

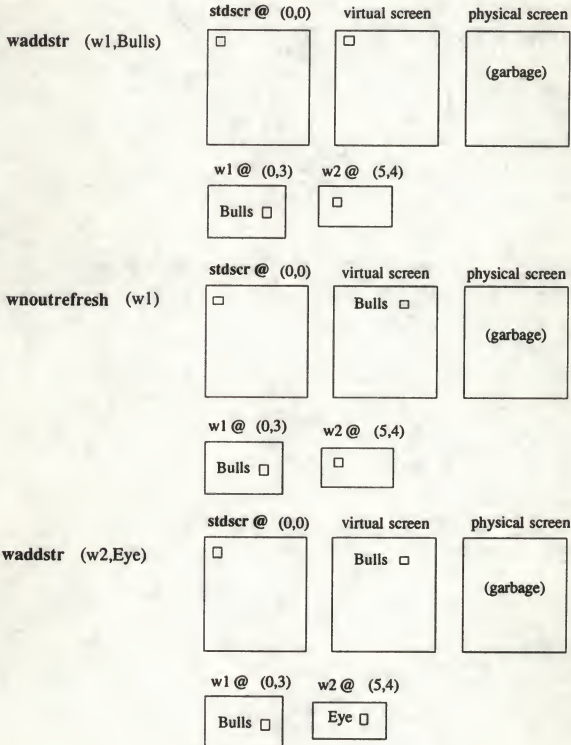
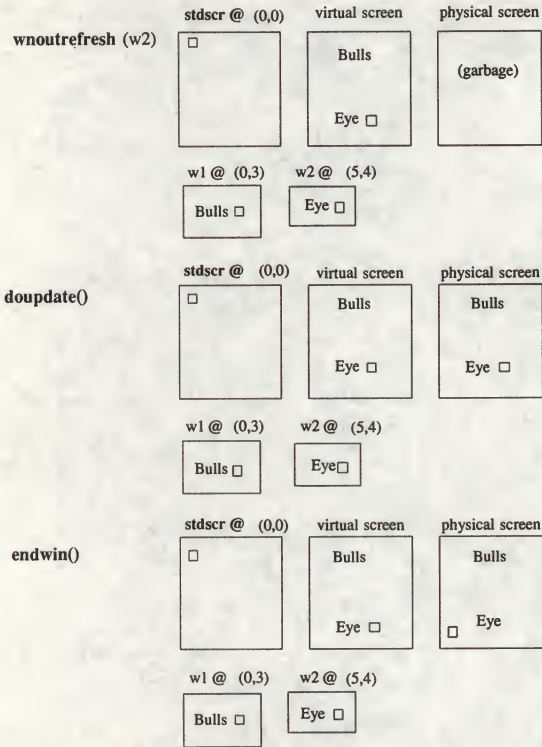


Figure 10-7 The Relationship Between a Window and a Terminal Screen
(continued)



New Windows

Following are descriptions of the routines **newwin()** and **subwin()**, which you use to create new windows. For information about creating new pads with **newpad()** and **subpad()**, see the **curses(S)** manual page.

newwin()

SYNTAX

```
#include <curses.h>
```

```
WINDOW *newwin(nlines, ncols, begin_y, begin_x)  
int nlines, ncols, begin_y, begin_x;
```

NOTES

- **newwin()** returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

Recall the sample program using two windows; see Figure 10-7. Also see the **window** program under “Program Examples” in this chapter.

Windows

subwin()

SYNTAX

```
#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

NOTES

- **subwin()** returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.

EXAMPLE

```
#include <curses.h>

main()
{
    WINDOW *sub;

    initscr();
    box(stdscr, 'w', 'w'); /* See the curses(S) manual page for box() */
    mvwaddstr(stdscr, 7, 10, "——— this is 10,10");
    mwaddch(stdscr, 8, 10, '|');
    mwaddch(stdscr, 9, 10, 'v');
    sub = subwin(stdscr, 10, 20, 10, 10);
    box(sub, 's', 's');
    wnoutrefresh(stdscr);
    wrefresh(sub);
    endwin();
}
```

This program prints a border of **w**'s around **stdscr** (the sides of your terminal screen) and a border of **s**'s around the subwindow **sub** when it is run. For another example, see the **window** program under "Program Examples" in this chapter.

ETI Low-Level Interface (curses) to High-Level Functions

In the following sections, we will consider the ETI high-level functions, which create and manipulate panels, menus, and forms. All application programs that use these high-level functions require a set of low-level ETI (**curses**) calls that properly initialize and terminate the programs. For convenience, you may want to isolate these calls in appropriate routines. Figure 10-8 shows one way you might do this. It lists routines to start low-level ETI, terminate it, and handle fatal errors.

Figure 10-8 Sample Routines for Low-Level ETI (**curses**) Interface

```
#include <curses.h>
static char * PGM    = (char *) 0; /* program name      */
static int   CURSES  = FALSE;      /* is curses initialized ? */

static void start_curses () /* curses initialization */
{
    CURSES = TRUE;
    initscr ();
    nonl ();
    raw ();
    noecho ();
    wclear (stdscr);
}

static void end_curses () /* curses termination */
{
    if (CURSES)
    {
        CURSES = FALSE;
        endwin ();
    }
}

static void error (f, s) /* fatal error handler */
char * f;
char * s;
{
    end_curses ();
    printf ("%s: ", PGM);
    printf (f, s);
    printf ("\n");
    exit (1);
}
```

These house-keeping routines use two global variables, **PGM** and **CURSES**. **PGM** is initialized with the program's name, while the Boolean **CURSES** is initialized with **FALSE** because **curses** itself has not yet been invoked.

Function **start_curses()** calls the low-level routines previously mentioned and sets **CURSES** to **TRUE** to indicate that it has initialized **curses**. Function **end_curses()** checks if **curses** is initialized and, if so, sets the variable **CURSES** to **FALSE** and terminates **curses**. The check is necessary because **endwin()** returns an error if called when **curses** is not initialized.

Function **error** is a universal fatal error handler—called whether or not **curses** is initialized. Function **error** first calls **end_curses()** to terminate the program if **curses** is on, and then prints the program's name (PGM) and the associated message. Finally, function **error** terminates the program itself using **exit()**.

Panels

Recall that a window is a rectangular area of the terminal screen on which you can write using the low-level ETI (**curses**) routines. You can create many windows on a screen, but if they overlap, portions of some windows intended to be hidden may nonetheless be visible when you use the low-level routines alone. To solve this problem, ETI uses the notion of a panel—a rectangle of text with depth.

Panels have depth only in relation to other panels and **stdscr**, which lies beneath all panels. The set of currently visible panels comprises the deck of panels.

Compiling and Linking Panel Programs

To use the panel routines, specify

```
#include <panel.h>
```

in your C program files and compile and link with the command line

```
cc [ flags ] files -lpanel -lcurses [ libraries ]
```

Creating Panels

This function creates a new panel on top of all existing panels in the deck. Its argument is a pointer to a window.

SYNTAX

```
PANEL *new_panel (window)
WINDOW      *window;    /* curses window to be
                        associated with new panel */
```

A pointer to the panel is returned if the panel is created; otherwise, the function returns NULL. The **new_panel()** operation fails if there is insufficient memory or if the window pointer argument is NULL. The window whose address is passed as an argument becomes associated with the panel. The size and location of the panel are the same as that of the low-level ETI (low-level ETI) window.

To create a panel, create a window, save the pointer to it, and use the pointer as an argument to **new_panel()**.

```
WINDOW *win;
PANEL *pptr;

win = newwin(2,6,0,3);
pptr = new_panel(win); /* after execution, pptr
                        stores pointer to new
                        panel */
```

Note that the newly created panel does not automatically have any adornments such as titles or borders. If you want your panel to have them, you must call appropriate low-level ETI routines with the panel's window as the argument.

When you create a new panel, it is automatically placed on top of the panel deck. Later, when you call **doupdate()** to adjust the visibility of all panels, the top panel is completely visible. On lower levels, a portion of a panel is visible only when no region of another panel is above it. Where two panels overlap, the higher one hides the lower. (The higher one is the newer one if neither has changed its position in the panel deck because of calls to **top_panel()**, **bottom_panel()**, or **show_panel()** described below.) If the panels do not overlap, the new panel is still logically above the old one. Their relative depth is not apparent until one of them is moved and overlaps the other.

Elementary Panel Window Operations

This section explains how you can fetch pointers to panel windows, change the windows associated with panels, and move panel windows to new locations on the screen.

Fetching Pointers to Panel Windows

Each panel has a low-level ETI window associated with it. To retrieve a pointer to this window, use function **panel_window()**.

SYNTAX

```
WINDOW *panel_window(panel)
PANEL *panel;           /* Panel whose window pointer
                        is returned */
```

The function returns NULL if the panel pointer argument is NULL.

In general, you may use this returned pointer as an argument to any standard low-level (**curses**) routine that takes a pointer to a window as an argument. For example, you can insert a character **c** at a location **y,x** in a panel window with the function **mvwinsch(win,y,x,c)**, where **win** is the window pointer returned by **panel_window()**.

```
WINDOW *win;
PANEL *panel;
int y, x;
chtype c;

win = panel_window(panel);
mvwinsch(win,y,x,c);
```

Elementary Panel Window Operations

Changing Panel Windows

To replace a panel's pointer to a window with a pointer to another window, call function **replace_panel()**. After the call, the panel remains at the same level within the panel deck.

SYNTAX

```
int replace_panel (panel, window)
PANEL *panel;      /* Panel with window to be replaced */
WINDOW *window;    /* New window pointer for panel */
```

This function returns OK if the operation is successful. If not, it returns ERR and leaves the original panel unchanged. Operation **replace_panel()** fails if the window pointer is NULL or there is insufficient memory.

To associate a panel with window **win1** and later replace its window by **win2**, you can write the following:

```
WINDOW *win1, win2;
PANEL *panel;

panel = new_panel(win1);

/* intervening processing with win1 as
   panel window */

replace_panel(panel, win2); /* change window
                             associated with
                             panel to win2 */
```

Once you have created additional windows with the low-level function **newwin()**, you in effect can reshape panel windows by using **replace_panel()**. To do so leaves the contents of the two windows unchanged.

Moving Panel Windows on the Screen

You should not move a panel's window by calling the low-level function `mvwin()` directly. To update the screen correctly, the panels subsystem must know the location of all panel windows, but function `mvwin()` does not inform the panels subsystem of the window's new location. To move a panel's window, you must call the function `move_panel()`, which moves a panel and its associated window and informs the panels subsystem of the move.

SYNTAX

```
int move_panel (panel, firstrow, firstcol)
PANEL *panel;      /* Panel to be moved */
int firstrow, firstcol; /* row/col of upper left
                        corner of new location
                        of window associated
                        with panel */
```

Note that the screen coordinates you specify are those for the upper left corner of the window in its new location. The panel may be moved to any location that the low-level ETI routines deem legitimate. In particular, a panel may be partly off the screen. The size, contents, and relative depth of the panel remain unchanged by `move_panel()`.

Function `move_panel()` returns OK if the operation was successful, ERR otherwise. The `move_panel()` operation fails if the low-level ETI functions are unable to move the panel's window, or if there is insufficient memory to satisfy the request. In these cases, the original panel remains unchanged.

To move the panel pointed to by `panel` such that its upper left corner is at row 22, column 45, you can write

```
PANEL *panel;

move_panel (panel, 22, 45);
```

Moving Panels to the Top or Bottom of the Deck

The relative depth of a panel can be changed by either pulling the panel to the top of the deck or by pushing it to the bottom. In either case, all other panels remain at the same depth relative to each other.

SYNTAX

```
int top_panel(panel)
PANEL *panel;

int bottom_panel(panel)
PANEL *panel;
```

Function **top_panel()** moves the panel pointed to by its argument to the top of the panel deck, while function **bottom_panel()** moves the panel to the bottom of the deck.

Both functions leave the size of the given panel, the contents of its associated window, and the relations of the other panels in the deck wholly intact. Both return OK if the operation is successful, ERR if not. The functions fail if the panel pointer argument is NULL or if the panel is hidden by a previous call to function **hide_panel()** described below.

To move the panel pointed to by **panel1** to the top of the deck of panels and the panel pointed to by **panel2** to the bottom of the deck, you can write the following:

```
PANEL * panel1, * panel2;

top_panel(panel1);
bottom_panel(panel2);
```

Updating Panels on the Screen

Function **update_panels()** makes all low-level **curses** calls (such as **touchwin()** and **wnoutrefresh()**) update all of the panels, so that proper depth relationships are maintained and only appropriate portions of panels are displayed.

SYNTAX

```
void update_panels();
```

The function does not, however, actually refresh your terminal screen. To do that, you must make a call to **doupdate()** whenever you want to display your latest changes.

To avoid displaying text on hidden panels, you should not use the low-level routines **wnoutrefresh()** and **wrefresh()** when working with panels.

Note

In general, do not use the low-level routines **wnoutrefresh()** or **wrefresh()** to display a window associated with a panel. Instead, use function **update_panels()** and function **doupdate()** to display the entire deck of panels.

If you use the low-level routines **wnoutrefresh()** or **wrefresh()** for a window associated with a panel, it will not be displayed properly, unless it happens to be associated with the top panel in the deck or is not hidden at all by other panel windows.

Recall that panels are always above **stdscr**, the standard ETI window. When a panel is moved or deleted, **stdscr** is updated along with the visible panels to ensure that it appears beneath all panels. Although **stdscr** has depth relative to other panels, it is not a panel, because panel operations like **top_panel()** and **bottom_panel()** do not apply. However, because **stdscr** rests beneath the deck of panels, you should always call **update_panels()** when you work with panels and change **stdscr**, even if you do not change any panels.

Updating Panels on the Screen

Function `wgetch()` automatically calls `wrefresh()`. Hence, if echo mode is active, when you request input from a window associated with a panel, be sure that the window is totally unobscured.

In summary, to update all panels and display them with their proper depth relationship, write:

```
WINDOW *win;

update_panels();
doupdate();
```

Finally, note that there is no way to display the updates to an obscured panel without displaying the changes to all panels.

Making Panels Invisible

ETI allows you to hide panels from the deck and later return them to it.

Hiding Panels

Panels may be temporarily hidden. This means that they are removed from the panel deck, but the memory allocated to them is not released.

SYNTAX

```
int hide_panel(panel)
PANEL *panel;    /* Pointer to panel to be hidden */
```

Hidden panels are not refreshed to the screen, but you may nonetheless apply nearly all panel operations to them.

Note

Only the operations **top_panel()**, **bottom_panel()**, and **hide_panel()** may not be applied to hidden panels because their panel arguments must belong to the deck of panels.

When you want to return a hidden panel to the deck of panels, use the function **show_panel()** described in the next section. When the panel is returned, it is placed on top of the deck.

To hide the panel pointed to by **panel2** above, write

```
PANEL *panel2;

hide_panel(panel2);
```

Function **hide_panel()** returns OK if the operation is successful and ERR if its panel pointer argument is NULL.

If you use function **hide_panel()** wisely, your program's performance can increase. You can hide a panel temporarily if no portion of it is to be displayed for awhile. An example is the hiding of a pop-up message.

Making Panels Invisible

Interim calls to function **update_panels()** will then execute faster. More importantly, you do not incur the overhead of creating the pop-up message.

Checking If Panels are Hidden

To enable you to check if a given panel is hidden, ETI provides the following function.

SYNTAX

```
int panel_hidden (panel)
PANEL * panel;
```

Function **panel_hidden()** returns a Boolean value (TRUE or FALSE) indicating whether or not its panel argument is hidden.

You might want to use this function before calling functions **top_panel()** or **bottom_panel()**, which do not operate on hidden panels. For example, to minimize the risk of having the error value ERR returned when moving a panel to the top of the deck, you can write

```
PANEL * panel;

if (! panel_hidden (panel)) /* panel in deck? */
    top_panel (panel); /* move panel to top of deck */
```


Reinstating Panels

This function is the opposite of function **hide_panel()**. It returns the hidden panel referenced in its argument to the top of the panel deck.

SYNTAX

```
int show_panel (panel)
PANEL *panel;      /*Panel to return to top of deck*/
```

Note that the panel must have been hidden by a previous **hide_panel()** call. The function returns OK if the operation is successful, and ERR if the panel pointer is NULL, if there is insufficient memory, or if the panel is not hidden.

For example, to return **panel2** to the deck, write

```
PANEL * panel2;

show_panel (panel2);
```

Fetching Panels Above or Below Given Panels

The following functions return a pointer to the panel immediately above or below the given panel. They are helpful in walking the panel deck from top to bottom or vice versa.

SYNTAX

```
PANEL *panel_above (panel)
PANEL *panel;      /* Get panel above this one */

PANEL *panel_below (panel)
PANEL *panel;      /* Get panel below this one */
```

Because hidden panels have no depth, they are excluded from these traversals.

Function **panel_above()** returns the panel immediately above the given panel. If its argument is **NULL**, it returns the bottommost panel. The function returns **NULL** if the given panel is on top or hidden, or if there are no visible panels.

Function **panel_below()** returns the panel immediately below the given panel. If its argument is **NULL**, it returns the topmost panel. The function returns **NULL** if the given panel is on the bottom of the deck of panels or hidden, or if there are no visible panels at all. There may be no visible panels at all for the following reasons:

- They have been hidden using **hide_panel()**
- All panels have been deleted
- No panels have been created.

Fetching Panels Above or Below Given Panels

If you want to do something to all panels or to search all of them for one with a particular attribute, you can place one of these functions in a loop. For example, to hide all panels (perhaps to display **stdscr** alone), you can write

```
{
  PANEL *panel, *pnl;

  for (panel=panel_above (NULL); panel; panel=panel_above(pnl))
  {
    pnl = panel;
    hide_panel(panel);
  }
}
```

Setting and Fetching the Panel User Pointer

To enable your application program to associate arbitrary data with a given panel, the ETI panel subsystem automatically allocates a pointer associated with each newly created panel. Initially, the value of this user pointer is NULL. You can set its value to whatever you want or not use it at all.

SYNTAX

```
int  *set_panel_userptr (panel, ptr)
PANEL *panel;           /* Panel whose user pointer to set */
char  *ptr;             /* user-defined pointer */

char *panel_userptr (panel)
PANEL *panel;           /* Panel whose user pointer to fetch */
char  *ptr;             /* user-defined pointer */
```

The user pointer has no meaning to the panels subsystem. Once the panel is created, the user pointer is neither changed nor accessed by the subsystem.

Function **set_panel_userptr()** sets the user pointer of a given panel to the value of your choice. The function returns OK if the operation is successful and ERR if the panel pointer is NULL.

Function **panel_userptr()** returns the user pointer for a given panel. If the panel pointer is NULL, the function returns NULL.

You can use these routines to store and retrieve a pointer to an arbitrary structure that holds information for your application. For example, you might use them to store a title or, as in Figure 10-9, create a hidden panel for pop-up messages.

Figure 10-9 Example Using Panel User Pointer

```
#include <curses.h>
#include <panel.h>

PANEL *msg_panel;
char *message = "Pop-up Message Here"; /* initialize message */

int display_deck (show_it)
int show_it;
{
    WINDOW *w;
    int rows, cols;

    if (show_it)
    {
        show_panel (msg_panel); /* reinstate panel */
        w = panel_window (msg_panel); /* fetch associated window */

        getmaxyx (w, rows, cols); /* fetch window size */

        /* center cursor */
        wmove (w, (rows-1), ((cols-1) - strlen(message))/2);

        /* fetch and write pop-up message */
        waddstr (w, panel_userptr (msg_panel));
    }
    update_panels(); /* display deck with message, if called for */
    doupdate();
    if (show_it)
        hide_panel (msg_panel); /* hide panel again, if necessary */
}

main()
{
    int show_mess = FALSE;

    msg_panel = new_panel (newwin (10, 10, 5, 60));
    set_panel_userptr (msg_panel, message); /*associate message
                                              with panel */
    hide_panel (msg_panel); /* remove from visible deck */

    /* if condition to display pop-up
       message is satisfied, set show_mess to TRUE */

    display_deck (show_mess);
}
```

Setting and Fetching the Panel User Pointer

After creating a window and its associated panel, **main()** calls **set_panel_userptr()** to set the panel user pointer to point to the panel's pop-up message string. Function **hide_panel()** hides the panel from the deck so that it is not normally displayed. Later, the application-defined routine **display_deck()** checks if the message is to be displayed. If so, it calls **show_panel()** which returns the panel to the deck and enables the panel to become visible on the next update and refresh. The message string returned by **panel_userptr()** is then written to the panel window. Finally, **update_panels()** adjusts the relative visibility of all panels in the deck, and **doupdate()** refreshes the screen. If called for, the pop-up message will now be visible.

Deleting Panels

The following function deletes a panel, but not its associated window. If you want to delete the window, you should use the low-level function `delwin()`.

SYNTAX

```
int del_panel (panel)
PANEL *panel;          /* Panel to be deleted */
```

The ETI panels subsystem assumes that the window associated with each panel always exists.

Note

If you want to delete a panel and its associated window, make sure that you delete the panel first, not the window. Your call to `del_panel()` should precede your call to `delwin()`.

However, it is not necessary to delete a window after its associated panel is deleted; if you like, you may associate the window with another panel.

Function `del_panel()` returns OK if the operation was successful, ERR otherwise. The `del_panel()` operation fails if the panel pointer is NULL.

To delete the panel referenced by **panel** and its associated window referenced by **win**, you can write

```
PANEL * panel;
WINDOW * win = panel_window(panel);

del_panel(panel);
delwin(win);
```

Menu

A menu is a screen display that presents a set of items from which the user selects one or more, depending on the type of menu. Once the user makes a selection, your application program responds accordingly. This response may be to generate a message, display another menu, or take some other action. Figure 10-10 displays a sample menu.

Figure 10-10 A Sample Menu

```
Black
Charcoal
Light Gray
Brown
Camel
Navy
Light Blue
Hunter Green
Gold
Burgundy
Rust
White

Choose an item
```

Compiling and Linking Menu Programs

To use the menu routines, specify

```
#include <menu.h>
```

in your C program files and compile and link with the command line

```
cc [ flags ] files -lmenu -lcurses [ libraries ]
```

If you use the panel routines as well, specify **-lpanel** before **-lcurses** on the command line.

Overview: Writing Menu Programs in ETI

This section introduces basic ETI menu terminology, lists the steps in a typical menu application program, and reviews the code in a simple example.

Some Important Menu Terminology

The following terms will be helpful:

item	a character string consisting of a name and an optional description
menu	a screen display that presents a set of items from which the user selects one or more, depending on the type of menu
connecting items to a menu	associating an array of item pointers with a menu
menu subwindow	a subwindow on which an associated menu is written
menu window	a window on which an associated menu subwindow and titles and borders, if any, are displayed
posting a menu	writing a menu on its associated subwindow
unposting a menu	erasing a menu from its associated subwindow
pattern matching	checking whether characters entered by the user match an item name of the menu

freeing a menu

deallocating the space for a menu and, as a byproduct, disconnecting an associated array of item pointers from a menu

freeing an item

deallocating the space for an item

NULL

generic term for a null pointer cast to the type of the particular object — item, menu, field, form, and so on

What a Menu Application Program Does

In general, a menu application program will do the following:

- initialize low-level ETI (**curses**)
- create the items for the menu
- create the menu
- post the menu
- refresh the screen
- process end user menu requests
- unpost the menu
- free the menu
- free items
- terminate low-level ETI (**curses**).

A Sample Menu Program

Figure 10-11 shows the ETI code necessary for generating the menu of colors in Figure 10-10.

Overview: Writing Menu Programs in ETI

Figure 10-11 Sample Menu Program to Create a Menu in ETI

```
#include <curses.h>
#include <menu.h>

char * colors[13] =
{
    "Black",          "Charcoal", "Light Gray",
    "Brown",          "Camel", "Navy",
    "Light Blue",     "Hunter Green", "Gold",
    "Burgundy",       "Rust", "White",
    (char *) 0
};

ITEM * items[13];

main ()
{
    MENU *      m;
    ITEM **     i = items;
    char *      c = colors;

    /* low-level ETI (curses) initialization */
    initscr ();
    nonl ();
    raw ();
    noecho ();
    wclear (stdscr);

    /* create items */
    while (*c)
        *i++ = new_item (*c++, "");
    *i = (ITEM *) 0;

    /* create and display menu */
    m = new_menu (i = items);
    post_menu (m);
    refresh ();
    sleep (5);

    /* erase menu and free both menu and items */
    unpost_menu (m);
    refresh ();
    free_menu (m);

    while (*i)
        free_item (*i++);

    /* low-level ETI (curses) termination */
    endwin ();
    exit (0);
}
```


Overview: Writing Menu Programs in ETI

To get an overview of ETI menu routines, we will now briefly walk through this menu program. In later sections, we discuss these and remaining ETI routines in detail.

Every menu program should have the line

```
#include <menu.h>
```

to instruct the C preprocessor to make the file of ETI menu declarations available. The initial low-level ETI routines establish the best terminal characteristics for working with the ETI menu routines.

The **while** loop creates each item for the menu using the ETI function **new_item()**. This function takes as its name argument a color from array **colors[]**. The optional description argument is here the null string. The new item pointers are assigned to a NULL-terminated array.

Next, the menu is created and the item pointer array is connected to the menu using function **new_menu()**. The menu is then posted to **stdscr** and the screen is refreshed to display the menu. The **sleep()** command makes the menu visible for 5 seconds.

To erase the menu, unpost it and refresh the screen. Function **free_menu** disconnects the menu from its item pointer array and deallocates the space for the menu. The last **while** loop uses function **free_item()** to free the space allocated for each item.

Finally, functions **endwin()** and **exit()** terminate low-level ETI and the program.

The following sections explain how to use all ETI menu routines. Program fragments illustrating the menu routines occur throughout this chapter. Many of these fragments are portions of a larger program example. The current example and others are included in the set of high-level ETI demonstration programs delivered with the ETI product. Low-level ETI demonstration programs are reproduced in the last section of this guide.

Note

Like all form routines that return an **int** value, all menu routines that do so return the value **E_OK** when they execute successfully.

Creating and Freeing Menu Items

Normally, to create a menu, you must first create the items comprising it. To create a menu item, use function **new_item()**.

SYNTAX

```
ITEM * new_item (name, description)
char * name;
char * description;
```

Function **new_item()** creates a new item by allocating space for the new item and initializing it. ETI displays the string **name** when the menu is later posted, but calling **new_item()** does not alone connect the item to a menu. The item **name** is also used in pattern-matching operations. If **name** is NULL or the null string, then **new_item()** returns NULL to indicate an error.

The argument **description** is a descriptive string associated with the item. It may or may not be displayed depending on the **O_SHOWDESC** option, which you can turn on or off with the **set_menu_opts()** and related functions described below. If **description** is NULL or the null string, no description is associated with the menu item.

If successful, **new_item()** returns a pointer to the new item. This pointer is the key to working with all item routines. When you pass it to them, it enables the menu subsystem to change, record, and examine the item's attributes.

If there is insufficient memory for the item, or **name** is NULL or the null string, then **new_item()** returns NULL.

In general, use an array to store the item pointers returned by **new_item()**. Figure 10-12 shows how you might create an item array of the planets of our solar system.

Figure 10-12 Creating an Array of Items

```
ITEM * planets[10];

planets[0] = new_item ("Mercury", "The first planet");
planets[1] = new_item ("Venus", "The second planet");
planets[2] = new_item ("Earth", "The third planet");
planets[3] = new_item ("Mars", "The forth planet");
planets[4] = new_item ("Jupiter", "The fifth planet");
planets[5] = new_item ("Saturn", "The sixth planet");
planets[6] = new_item ("Uranus", "The seventh planet");
planets[7] = new_item ("Neptune", "The eighth planet");
planets[8] = new_item ("Pluto", "The ninth planet");
planets[9] = (ITEM *) 0;
```

Function **new_item()** does not copy the name or description strings, but saves the pointers to them. So once you call **new_item()**, you should not change the strings until you call **free_item()**.

SYNTAX

```
free_item(item);
ITEM * item;
```

Function **free_item()** frees an item. It does not, however, deallocate the space for the item's name or description.

The argument to **free_item()** is a pointer previously obtained from **new_item()**.

Note

To free an item, you must have already created it with **new_item()** and it must not be connected to a menu. If these conditions are not met, **free_item()** returns one of the error values listed below.

Once an item is freed, you must not use it again. If a freed item's pointer is passed to an ETI routine, undefined results will occur.

If successful, **free_item()** returns E_OK. If it encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null item
E_CONNECTED	- item is connected to a menu

Two Kinds of Menus: Single- and Multi-Valued

Menus are of two kinds:

single-valued menus	from which the user may select only one item
multi-valued menus	from which the user may select one or more items

By default, every menu is single-valued. To create a multi-valued menu, you turn off menu option `O_ONEVALUE` using function `set_menu_opts()` or `menu_opts_off()`. These functions are treated in the section "Setting Item Options."

Menus of both types always have a current item. With single-valued menus, you determine the item selected by noting the current item. With multi-valued menus, you determine all items selected by applying function `item_value()` to each menu item and noting the value returned. Most menu functions pertain to menus whether they are single- or multi-valued. Function `set_item_value()`, however, may be used only with multi-valued menus.

Manipulating an Item's Select Value in a Multi-Valued Menu

Select values of an item are either `TRUE` (selected) or `FALSE` (not selected). Function `set_item_value()` sets the select value of an item, while `item_value()` returns it.

SYNTAX

```
int set_item_value (item, value)
ITEM * item;
int value;

int item_value (item)
ITEM * item;
```

Two Kinds of Menus: Single- and Multi-Valued

Function `set_item_value()` fails if given an item that is not selectable (the `O_SELECTABLE` option was previously turned off) or the item is connected to a single-valued menu (connecting items to menus is described in the section, "Creating and Freeing Menus"). If successful, `set_item_value()` returns `E_OK`. Otherwise, one of the following is returned.

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_REQUEST_DENIED</code>	- item not selectable or single value menu

If the argument to `item_value()` is an item pointer connected to a single-valued menu, `item_value()` returns `FALSE`.

You might want to place the code in Figure 10-13 after your user responds to a menu. Function `process_menu()` determines which items have been selected, processes them appropriately, and marks them as unselected to prepare for further user response.

Figure 10-13 Using `item_value()` in Menu Processing

```
void process_menu (m) /* process multi-valued menu */
MENU * m;
{
    ITEM ** i = menu_items (m);

    while (*i)
    {
        if (item_value (*i))
        {
            /* take action appropriate for selection
               of this item */

            set_item_value (*i, FALSE);
        }
        ++i;
    }
}
```

Manipulating Item Attributes

An attribute is any feature whose value can be set or read by an appropriate ETI function. An item attribute is any item feature whose value can be set or read by an appropriate ETI function. Item names, descriptions, options, and visibility are examples of item attributes.

Fetching Item Names and Descriptions

The routines `item_name()` and `item_description()` take an item pointer as their argument. Function `item_name()` returns the item's name, while function `item_description()` returns its description.

SYNTAX

```
char * item_name (item)
ITEM * item;

char * item_description (item)
ITEM * item;
```

Both functions return NULL if given a NULL item pointer.

Setting Item Options

An option is an attribute whose value may be either on or off. The current release of ETI provides the item option `O_SELECTABLE`. (In the future, ETI may provide additional options.) Setting the `O_SELECTABLE` option lets your user select the item. By default, `O_SELECTABLE` is set for every item. Function `set_item_opts()` lets you turn on or turn off this and any future options for an item, while `item_opts()` lets you examine the option(s) set for a given item.

SYNTAX

```
int set_item_opts (item, opts)
ITEM * item;
OPTIONS opts;

OPTIONS item_opts (item)
ITEM * item;

options:
    O_SELECTABLE
```


In addition to turning on the named item options, function **set_item_opts()** turns off any other item options.

If successful, **set_item_opts()** returns **E_OK**. Otherwise, it returns the following:

E_SYSTEM_ERROR	- system error
-----------------------	----------------

If function **set_item_opts()** is passed a **NULL** item pointer, like other functions it sets the new current default. If function **item_opts()** is passed a **NULL** pointer, it returns the current default.

If you turn off option **O_SELECTABLE**, the item cannot be selected. You might want to make an item unselectable to emphasize certain things your application program is doing. Unselectable items are displayed using the grey display attribute, described below in the section "Setting Menu Display Attributes."

Because options are Boolean values (they are either on or off), you must use C Boolean operators with **item_opts()** to turn them on and off. Consequently, to turn off option **O_SELECTABLE** for item **i0** and turn on the same option for item **i1**, write the following:

```
ITEM * i0, * i1;
```

```
set_item_opts (i0, item_opts (i0) & ~O_SELECTABLE); /*turn option off */  
set_item_opts (i1, item_opts (i1) | O_SELECTABLE); /*turn option on */
```

ETI also enables you to turn on and off specific item options without affecting others, if any. The following functions change only the options specified.

SYNTAX

```
int item_opts_on (item, opts)  
ITEM * item;  
OPTIONS opts;  
  
int item_opts_off (item, opts)  
ITEM * item;  
OPTIONS opts;
```

Manipulating Item Attributes

These functions return the same error conditions as `set_item_opts()`.

For example, the following code turns option `O_SELECTABLE` off for item `i0` and on for item `i1`.

```
ITEM * i0, * i1;

item_opts_off (i0, O_SELECTABLE); /* turn option off */

item_opts_on  (i1, O_SELECTABLE); /* turn option on  */
```

To change the current default to not `O_SELECTABLE`, you can write either

```
/* set current defaults for all new items */

set_item_opts ((ITEM*) 0, item_opts( (ITEM*) 0) & ~O_SELECTABLE);

or

item_opts_off ((ITEM*) 0, O_SELECTABLE); /* turn default option off */
```

Checking an Item's Visibility

A menu item is visible if it appears in the subwindow of the posted menu to which it is connected. (Connecting and Posting Menus is described below.) Function `item_visible()` enables your application program to determine if an item is visible.

SYNTAX

```
int item_visible (item)
ITEM * item;
```

If the item is connected to a posted menu and it appears in the menu subwindow, `item_visible()` returns `TRUE`. Otherwise, it returns `FALSE`.

To check if the first menu item is currently visible on the display, write

```
int at_top (m) /*check visibility of first menu item*/
MENU * m;
{
    ITEM ** i = menu_items (m);
    ITEM * firstitem = i[0];

    return item_visible (firstitem);
}
```

For another example, see the section “Counting the Number of Menu Items.”

Changing the Current Default Values for Item Attributes

ETI establishes initial current default values for item attributes. During item initialization, each item attribute is assigned the current default value of the attribute. You can change or retrieve the current default attribute values by calling the appropriate function with a NULL item pointer. After the current default value changes, all subsequent items created with `new_item()` will have the new default value.

Note

Items created before changing the current default value retain their previously assigned values.

The following sections offer many examples of how to change item attributes.

Setting the Item User Pointer

For each item created, ETI automatically allocates a special user pointer that enables you to associate arbitrary data with the item. By default, the user pointer's value is NULL. You may set its value to whatever you want or not use it at all.

SYNTAX

```
int set_item_userptr (item, userptr)
ITEM * item;
char * userptr;

char * item_userptr (item)
ITEM * item;
```

These two functions are helpful for creating item data such as title strings, help messages, and the like.

Any defined structure can be connected to an item using the item's user pointer. The pointer must be cast to (char *) and then later recast back to (defined-struct *). Figure 10-14 shows how to use an item's user pointer with a struct ITEM_ID, which stores biological information.

Figure 10-14 Using an Item User Pointer

```

typedef struct
{
    int      id;
    char *   name;
    char *   type;
}
    ITEM_ID;

ITEM_ID ids[7] =
{
    1, "apple", "fruit",
    2, "ant", "insect",
    3, "cow", "mammal",
    4, "lizard", "reptile",
    5, "potato", "vegetable",
    6, "zebra", "mammal",
    0, "", "",
};

ITEM * items[7];

for (i = 0; ids[i]; ++i)
{
    /* create item from each ids.name */

    items[i] = new_item (ids[i].name, "");

    /*set userpointer to point to start of each struct in ids[i]*/

    set_item_userptr (items[i], (char *) &ids[i]);
}
items[i] = (ITEM *) 0;

```

Note that the pointer to each entry in array **ids** is cast to **char ***, which **set_userptr()** requires. You might then write a function that uses function **item_userptr()** to return the information. The following function returns the item type.

```

char * get_type (i)
ITEM * i;
{
    ITEM_ID * id = (ITEM_ID *) item_userptr (i);
    return id -> type;
}

```

Here, the value returned by **item_userptr()** is recast to **ITEM_ID *** so the item's **type** may be found.

Setting the Item User Pointer

Finally, you might call `get_type()` to write the type, thus:

```
WINDOW * win;  
  
waddstr (win, get_type(i));
```

If successful, `set_item_userptr()` returns `E_OK`. Otherwise, it returns the following:

<code>E_SYSTEM_ERROR</code>	- system error
-----------------------------	----------------

If function `set_item_userptr()` is passed a `NULL` item pointer, the argument `userptr` becomes the new default user pointer for all subsequently created items. For example, the following sets the new default user pointer to point to the string "You are Here":

```
set_item_userptr( (ITEM *) 0, "You are Here");
```

Creating and Freeing Menus

Once you create the items for your menu, you can create the menu. To create and initialize a menu, use function **new_menu()**.

SYNTAX

```
MENU * new_menu (items)
ITEM ** items;
```

The argument to **new_menu()** is a NULL terminated, ordered array of ITEM pointers. These pointers define the items on the menu. Their order determines the order in which the items are visited during menu driver processing, described below.

Function **new_menu()** does not copy the array of item pointers. Instead, it saves the pointer to the array for future use.

Note

Once your application program has called **new_menu()**, it should not change the array of item pointers until the menu is freed by **free_menu()** or the item array is replaced by **set_menu_items()**, as described below.

Items passed to **new_menu()** are connected to the menu created. They cannot be simultaneously connected to another menu. To disconnect the items from a menu, you can use function **free_menu()** or function **set_menu_items()**, which changes the items connected to a menu from one set to another. See the section “Fetching and Changing Menu Items.”

If successful, **new_menu()** returns a pointer to the new menu. The following error conditions hold:

- If there is insufficient memory for the menu or it detects an item connected to another menu, **new_menu()** returns NULL.
- If the array of item pointers is not NULL-terminated, undefined results occur.

Creating and Freeing Menus

In addition, if **new_menu()**'s argument **items** is **NULL**, as in

```
MENU * m;  
  
m = new_menu ((MENU *) 0);
```

it creates the menu with no items connected to it and assigns the menu pointer to **m**.

The menu pointer returned by **new_menu()** is the key to working with all menu routines. Pass it to the appropriate menu routine to do such tasks as post menus, call the menu driver, set the current item, and record or examine menu attributes.

Turn again to Figure 10-11 for an example of how to create a menu. In general, you want to use a **while** loop as illustrated to create the menu items and assign the item pointers to the item pointer array. Note the **NULL** terminator assigned to the item pointer array before the menu is created with **new_menu()**.

When you no longer need a menu, you should free the space allocated for it. To do this, use function **free_menu()**.

SYNTAX

```
int free_menu (menu)  
MENU * menu;
```

Function **free_menu()** takes as its argument a menu pointer previously obtained from **new_menu()**. It disconnects all items from the menu and frees the space allocated for the menu. The items associated with the menu are not freed, however, because you may want to connect them to another menu. If not, you can free them by calling **free_item()**.

Remember that once a menu is freed, you must not pass its menu pointer to another routine. If you do, undefined results occur.

If successful, calls to **free_menu()** return **E_OK**. If **free_menu()** encounters an error, it returns one of the following:

E_BAD_ARGUMENT	- NULL menu pointer
E_POSTED	- menu is posted
E_SYSTEM_ERROR	- system error

For **E_POSTED**, see the section "Posting and Unposting Menus."

Manipulating Menu Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A menu attribute is any menu feature whose value can be set or read by an appropriate ETI function. The set of items connected to a menu and the number of items in the menu are examples of menu attributes.

Fetching and Changing Menu Items

During processing, you may sometimes want to change the set of items connected to a menu. Function `set_menu_items()` enables you to do this.

SYNTAX

```
int set_menu_items (menu, items)
MENU * menu;
ITEM ** items;

ITEM ** menu_items (menu)
MENU * menu;
```

Like the argument to `new_menu()`, the second argument to `set_menu_items()` is a NULL-terminated, ordered array of ITEM pointers that defines the items on the menu. Like `new_menu()`, function `set_menu_items()` does not copy the array of item pointers. Instead, it saves the pointer to the array for future use.

The items previously connected to the given menu when `set_menu_items()` is called are disconnected from the menu (but not freed) before the new items are connected. The new items cannot be given to other menus unless first disconnected by `free_menu()` or another `set_menu_items()` call.

If `items` is NULL, the items associated with the given menu are disconnected from it, but no new items are connected.

Manipulating Menu Attributes

If function `set_menu_items()` is successful, it returns `E_OK`. If it encounters an error, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- NULL menu pointer or NULL associated item array
<code>E_POSTED</code>	- menu is posted
<code>E_CONNECTED</code>	- connected item

Function `menu_items()` returns the array of item pointers associated with its menu argument. In the next section, the application-defined function `at_bottom()` illustrates its use.

If no items are connected to the menu or the menu pointer argument is NULL, `menu_items()` returns NULL.

As an example of `set_menu_items()`, consider Figure 10-15, whose code changes the items associated with a previously created menu.

Figure 10-15 Changing the Items Associated With a Menu

```
MENU *m;
ITEMS ** olditems, ** newitems;
/* create items */

m = new_menu(olditems); /* create menu m */
/* process menu with olditems */

set_menu_items (m,newitems); /* change items associated with menu m */
```

Counting the Number of Menu Items

Occasionally, you may want to do different processing, depending on the number of items connected to your current menu. Function `item_count()` returns the number of items connected to a menu.

```
SYNTAX

int item_count (menu)
MENU * menu;
```

If `menu` is NULL, function `item_count()` returns -1.

As an example of the use of this function, consider the following routine. Because the index to the last menu item is one less than the number of items, this routine determines whether the last item is displayed.

```
int at_bottom (m) /* check visibility of last menu item */
MENU * m;
{
    ITEM ** i = menu_items (m);
    ITEM * lastitem = i[item_count(m)-1];

    return item_visible (lastitem);
}
```

Changing the Current Default Values for Menu Attributes

As it does with the attributes of other objects, ETI establishes initial current default values for menu attributes. During menu creation, each menu attribute is assigned the current default value of the attribute. You can change or retrieve the current default attribute values by calling the appropriate function with a NULL menu pointer. After the current default value changes, all subsequent menus created with `new_menu()` will have the new default value.

Note

Menus created before changing the current default value retain their previously assigned values.

The following sections offer many examples of how to change menu attributes.

Displaying Menus

In general, to display a menu, you must determine the menu's dimensions, optionally associate a window and subwindow with the menu, optionally set the menu's display attributes, post the menu, and refresh the screen.

Determining the Dimensions of Menus

The simplest way to display a menu is to use **stdscr** as your default window and subwindow. Any titles, borders, or other decorative matter are displayed in the menu window; the menu proper is displayed in the menu subwindow. If you want to specify a menu window or subwindow, use the functions **set_menu_win()** or **set_menu_sub()**. (These routines are treated in the section "Associating Windows and Subwindows with Menus.") Whether or not you choose a menu window, ETI calculates the minimum window (or subwindow) size for your menu.

To determine the minimum window size for a menu, ETI considers five factors:

- the size and number of items in a menu
- whether option **O_ROWMAJOR** is on
- whether option **O_SHOWDESC** is on
- the format, or maximum number of rows and columns on a displayed page of the menu
- the mark string for menu items

ETI knows the size and number of items in a menu as soon as you call **new_menu()**, discussed above. By default, options **O_ROWMAJOR** and **O_SHOWDESC** are on. Option **O_ROWMAJOR** ensures that the items are displayed in row major order — fanning out left to right, then top to bottom. Option **O_SHOWDESC** ensures that an item's description, if any, is displayed with the item's name.

This section first describes the menu's format and mark string. It then describes the routine **scale_menu()**, which uses the above information to set the window size for the menu.

Note

The five factors that determine the minimum window size have default values. You need not worry about them until you want to customize your menus.

Specifying the Menu Format

In general, the items comprising a menu do not fill a single screen. Sometimes they occupy considerably less space, sometimes considerably more. The following functions enable you to set the maximum number of rows and columns of menu items to be displayed at any one time.

SYNTAX

```
int set_menu_format (menu, maxrows, maxcols)
MENU * menu;
int maxrows, maxcols;

void menu_format (menu, maxrows, maxcols)
MENU * menu;
int * maxrows, * maxcols;
```

A menu page is the collection of currently visible items. Function **set_menu_format()** establishes the maximum number of rows and columns of items that may be displayed on a menu page.

The actual number of rows and columns displayed may be less than **maxrows** or **maxcols** depending on the number of items and whether the **O_ROWMAJOR** option is on. (Menu options are described in the section, “Setting and Fetching Menu Options”.) Function **menu_format()** returns the maximum number of rows and columns of items that you set for the given menu.

The default number of item rows is 16, while the default number of item columns is 1. If either **maxrows** or **maxcols** equals 0 in the call to **set_menu_format()**, the current value is not changed. An error occurs, however, if the value of either of these arguments is less than 0.

Displaying Menus

ETI calculates the total number of rows and columns in a row major menu as follows:

```
#define minimum(a,b) ((a) < (b) ? (a) : (b))

total_rows = (number_of_items - 1) / maxcols + 1;
total_cols = minimum (number_of_items, maxcols);
```

ETI calculates the total number of rows and columns in a column major menu as follows:

```
total_rows = (number_of_items - 1) / maxcols + 1;
total_cols = (number_of_items - 1) / total_rows + 1;
```

Whether or not the `O_ROW_MAJOR` option is on, the number of rows and columns of items that are displayed at one time on a menu page is

```
displayed_rows = minimum (total_rows, maxrows);
displayed_cols = minimum (total_cols, maxcols);
```

If **total_rows** is greater than **maxrows**, the menu is scrollable — your end-user can scroll up or down through the menu by making the appropriate menu driver request. See the section “Menu ETI Requests.”

As an example, consider the displays in Figures 10-16 and 10-17. They portray menus consisting of 5 items. The numbers 0 through 4 signify menu items in the order in which they live in the item pointer array. Figure 10-16 shows the menu displayed with a format of maximum number of rows 2, maximum number of columns 2. To stipulate this format for menu **m**, write

```
set_menu_format (m, 2, 2);
```

Using the formulas above, we see that **total_rows** is 3 and **total_cols** is 2 in all four cases displayed in the two figures. The first display in each figure shows the menu in row-major format (`O_ROW_MAJOR` on), the second in column-major format. The displayed number of rows and columns in Figure 10-16 is 2. To see the last row of items, your user can make the `REQ_SCR_DLINE` request to scroll down. If, instead, you set the format of this menu to 3 rows, 2 columns, you get 1 of the 2 displays in Figure 10-17. The enclosing block in each case indicates the items displayed at one time.

Figure 10-16 Examples of Menu Format (2, 2)

0	1
2	3

4

Row Major

0	3
1	4

2

Column Major

Maximum Rows 2

Figure 10-17 Examples of Menu Format (3, 2)

0	1
2	3
4	

Row Major

0	3
1	4
2	

Column Major

Maximum Rows 3

For a larger example, consider Figure 10-18. Here the number of items is 18 and the format in both cases is four rows, three columns. In both cases, the total number of rows comes to 6, the total number of columns to 3, and the displayed number of rows to 4. Calculation shows that changing the number of items in this example to 19 changes the number of rows to 7.

Displaying Menus

Figure 10-18 Examples of Menu Format (4, 3)

0	1	2
3	4	5
6	7	8
9	10	11

12 13 14
15 16 17

Row Major

0	6	12
1	7	13
2	8	14
3	9	15

4 10 16
5 11 17

Column Major

The column major examples emphasize that when the total number of rows is greater than the maximum number of rows, the items displayed do not exactly follow the order of the items in the array of item pointers. The items are arranged in column-major format throughout the entire menu, not within each displayed page. This conception agrees with your user's ability to scroll through the menu.

If successful, function `set_menu_format()` returns `E_OK`. If an error occurs, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- rows < 0 or cols < 0
<code>E_POSTED</code>	- menu is posted

If function `set_menu_format()` is passed a `NULL` menu pointer, it sets a new system default. Suppose, for instance, that you want to change the default maximum number of rows of items displayed to 10, and the default maximum number of columns displayed to 3. You can write

```
set_menu_format((MENU *)0,10,3);
```

The function `set_menu_format()` resets the value of `top_row()` to 0. See the section "Fetching and Changing the Top Row" for details.

Finally, if function `menu_format()` receives a `NULL` menu pointer, it returns the current default format.

Changing Your Menu's Mark String

The mark string distinguishes

- selected items in a multi-valued menu
- the current item in a single-valued menu

The mark string appears just to the left of the item name.

SYNTAX

```
int set_menu_mark (menu, mark)
MENU * menu;
char * mark;

char * menu_mark (menu)
MENU * menu;
```

Function **set_menu_mark()** sets the mark string, while **menu_mark()** returns the string. The initial default mark string is a minus sign (“-”). The mark string may be as long as you want, provided each item fits on one line of the menu’s subwindow.

Note

Do not change the mark string area as long as you want that mark, because ETI does not copy it.

If **mark** is NULL, no mark string appears.

You can call **set_menu_mark()** either before or after the menu is posted. (See the section “Posting and Unposting Menus.”) However, there is a restriction to calling it afterwards.

Note

If you call **set_menu_mark()** with a posted menu, the length of the mark string must stay the same.

If the menu is posted and the length of the mark string changes, the function returns **E_BAD_ARGUMENT** and leaves the mark unchanged.

Displaying Menus

To change the mark string for menu **m** to "--->", you can write

```
MENU * m;  
  
set_menu_mark (m, "--->"); /* change mark string for menu m */
```

If successful, function **set_menu_mark()** returns **E_OK**. If an error occurs, function **set_menu_mark()** returns one of the following:

```
E_SYSTEM_ERROR - system error  
E_BAD_ARGUMENT - menu is posted: change  
                  in string length impossible
```

Note that you can change the current default mark string for all subsequently created menus in your program by passing **set_menu_mark()** a **NULL** menu pointer. To change the current default mark string to "--->", write

```
set_menu_mark ((MENU *) 0, "--->"); /* change default mark string */
```

All subsequently created menus will have "--->" as their mark string. To return the current default mark string, call **menu_mark()** with **NULL**:

```
char * mark = menu_mark ((MENU *) 0); /* default mark string */
```

Querying the Menu Dimensions

Remember that the size of menu items, the **O_ROWMAJOR** menu option, the menu format, and the menu mark determine the smallest window size for a menu. Function **scale_menu()** returns this smallest window size in terms of the number of character rows and columns.

SYNTAX

```
int scale_menu (menu, rows, cols)  
MENU * menu;  
int * rows, * cols;
```

Because function **scale_menu()** must return more than one value (namely, the minimum number of rows and columns for the menu) and C passes parameters "by value" only, the arguments of **scale_menu()** are pointers. The pointer arguments **rows** and **cols** point to locations used to return the minimum number of rows and columns for displaying the menu.

Note

You should call `scale_menu()` only after the menu's items have been connected to the menu using `new_menu()` or `set_menu_items()`.

The following code places the minimal number of rows and columns necessary for menu `m` in `rows` and `cols`:

```
MENU *m;
int rows, cols;

scale_menu (m, &rows, &cols); /* return dimensions of menu m */
```

You use the values returned from `scale_menu()` to create menu windows and subwindows. In the next section, we will see how to do this.

If successful, `scale_menu()` returns `E_OK`. If an error occurs, the function returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu pointer
<code>E_NOT_CONNECTED</code>	- no connected items

Associating Windows and Subwindows with Menus

Two windows are associated with each menu—the menu window and the menu subwindow. The following functions assign windows and subwindows to menus and fetch those previously assigned to them.

SYNTAX

```
int set_menu_win (menu, window)
MENU * menu;
WINDOW * window;

WINDOW * menu_win (menu)
MENU * menu;

int set_menu_sub (menu, window)
MENU * menu;
WINDOW * window;

WINDOW * menu_sub (menu)
MENU * menu;
```

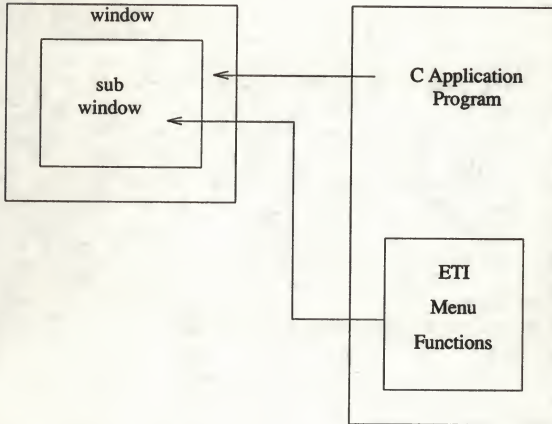
To place a border around your menu or give it a title, call `set_menu_win()` and write to the associated window.

Note

By default, (C) the menu window is NULL, which by convention means that ETI uses `stdscr` as the menu window; and (S) the menu subwindow is NULL, which means that ETI uses the menu window as the menu subwindow.

If you do not want to use the system defaults, you may create a window and a subwindow for every menu. ETI automatically writes all output of the menu proper on the menu's subwindow. You may write additional output (such as borders, titles, and the like) on the menu's window. The relationship between ETI menu routines, your application program, a menu window, and a menu subwindow is illustrated in Figure 10-19.

Figure 10-19 Menu Functions Write to Subwindow, Application to Window



Note

You should apply all output and refresh operations to the menu window, not its subwindow.

Figure 10-20 shows how you can create and display a menu with a border of the default characters, `ACS_VLINE` and `ACS_HLINE`. (See the entry on the `box` command in the `curses(S)` manual page.)

Displaying Menus

Figure 10-20 Creating a Menu with a Border

```
MENU * m;
WINDOW * w;
int rows, cols;

scale_menu (m, &rows, &cols); /* get dimensions of menu */

/* create window 2 characters larger than menu dimensions
with top left corner at (0, 0). subwindow is positioned
at (1, 1) relative to menu window origin with dimensions
equal to the menu dimensions. */

if (w = newwin (rows+2, cols+2, 0, 0))
{
    set_menu_win (m, w);
    set_menu_sub (m, derwin (w, rows, cols, 1, 1));

    box (w, 0, 0); /* draw border in w */
}
```

Variables **rows** and **cols** provide the menu dimensions without the border. The dimensions of the menu subwindow are set to these values. In general, if you want a simple border, you should set the number of rows and columns in the menu's window to be two more than the numbers in its subwindow, as in the example.

Remember that the initial default menu window and subwindow are NULL. (By convention, this means that **stdscr** is used as the menu window and the menu window is used as the menu subwindow.) If you want to change the current default menu window or subwindow, you can pass functions **set_menu_win()** and **set_menu_sub()** a NULL menu pointer. Thus, the code

```
WINDOW * dftwin;

set_menu_win ((MENU *) 0, dftwin); /* sets default menu
                                     window to dftwin */
```

changes the current default window to **dftwin**.

If successful, functions **set_menu_win()** and **set_menu_sub()** return **E_OK**. If not, they return one of the following:

E_SYSTEM_ERROR	- system error
E_POSTED	- menu is posted

Fetching and Changing A Menu's Display Attributes

Menu display attributes are visible menu characteristics that distinguish classes of menu items from each other. Low-level ETI (**curses**) color and video attributes are used to differentiate the menu display attributes. These menu display attributes include

foreground attribute	distinguishes the current item, if selectable, on all menus and selected items on multi-valued menus
background attribute	distinguishes selectable, but unselected, items on all menus
grey attribute	distinguishes unselectable items on multi-valued menus
pad character	the character that fills (pads) the space between a menu item's name and description

Displaying Menus

The following functions enable you to set and read these attributes.

SYNTAX

```
int set_menu_fore (menu, attr)
MENU * menu;
chtype attr;
```

```
chtype menu_fore (menu)
MENU * menu;
```

```
int set_menu_back (menu, attr)
MENU * menu;
chtype attr;
```

```
chtype menu_back (menu)
MENU * menu;
```

```
int set_menu_grey (menu, attr)
MENU * menu;
chtype attr;
```

```
chtype menu_grey (menu)
MENU * menu;
```

```
int set_menu_pad (menu, pad)
MENU * menu;
int pad;
```

```
int menu_pad (menu)
MENU * menu;
```

In general, to establish uniformity throughout your program, you should set the menu display attributes with these functions at the start of the program.

Function **set_menu_fore()** sets the **curses** foreground attribute. The default is **A_STANDOUT**.

Function **set_menu_back()** sets the **curses** background attribute. The default is **A_NORMAL**.

Function **set_menu_grey()** sets the **curses** attribute used to display non-selectable items. The default is **A_UNDERLINE**.

To set the foreground attribute of menu **m** to **A_BOLD** and its background attribute to **A_DIM**, you write

```
MENU *m;

set_menu_fore(m, A_BOLD);
set_menu_back(m, A_DIM);
```

All these functions can change or fetch the current default if passed a **NULL** menu pointer. For example, to set the default grey attribute to **A_NORMAL**, write

```
set_menu_grey((MENU *)0, A_NORMAL);
```

If functions **set_menu_fore()**, **set_menu_back()**, and **set_menu_grey()** encounter an error, they return one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- bad curses attribute

Function **set_menu_pad()** sets the pad character for a menu. The initial default pad character is a blank. The pad character must be a printable ASCII character.

To change the pad character for menu **m** to a dot ('.'), write

```
MENU * m;

set_menu_pad(m, '.');
```

If function **set_menu_pad()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- nonprintable pad character

Posting and Unposting Menus

To post a menu is to write it on the menu's subwindow. To unpost a menu is to erase it from the menu's subwindow, but not destroy its internal data structure. ETI provides two routines for these actions.

Displaying Menus

SYNTAX

```
int post_menu (menu)
MENU * menu;

int unpost_menu (menu)
MENU * menu;
```

Note that neither of these functions actually change what is displayed on the terminal. After posting or unposting a menu, you must call **wrefresh()** (or its equivalents, **wnoutrefresh()** and **doupdate()**) to do so.

If function **post_menu()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_POSTED	- menu is already posted
E_NOT_CONNECTED	- no connected items
E_NO_ROOM	- menu does not fit in subwindow

Regarding **E_NO_ROOM**, recall from the section “Querying the Menu Dimensions” that function **scale_menu()** returns the number of rows and columns necessary to display the menu. It does not, however, know the size of the subwindow you are associating with the menu. Only when the menu is posted is this point checked. Any failure of the menu to fit in the subwindow is then detected.

If function **unpost_menu()** executes successfully, it returns **E_OK**. In the following situations, it fails and returns the indicated values:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_NOT_POSTED	- menu is not posted
E_BAD_STATE	- called from init or term

You might, for instance, receive **E_NOT_POSTED** if you forgot to post the menu in the first place or you mistakenly tried to unpost it twice.

Figure 10-21 illustrates two routines you might write to post and unpost menus. Function **display_menu()** creates the window and subwindow for the menu and posts it. Most of its code we saw previously in Figure 10-20. Function **erase_menu()** unposts the menu and erases its associated window and subwindow.

Figure 10-21 Sample Routines Displaying and Erasing Menus

```
static void display_menu (m)    /* create menu windows and post */
MENU * m;
{
    WINDOW * w;
    int rows;
    int cols;

    scale_menu (m, &rows, &cols);    /* get dimensions of menu */
    /* create menu window, subwindow, and border */
    if (w = newwin (rows+2, cols+2, 0, 0)) {
        set_menu_win (m, w);
        set_menu_sub (m, derwin (w, rows, cols, 1, 1));
        box (w, 0, 0);    /* create border of 0's */
        keypad (w, 1);    /* set for each data entry window */
    }
    else
        error ("error return from newwin", NULL);

    /* post menu */

    if (post_menu (m) != E_OK)
        error ("error return from post_menu", NULL);
    else
        wrefresh (w);
}

static void erase_menu (m)    /* unpost and delete menu windows */
MENU * m;
{
    WINDOW * w = menu_win (m);
    WINDOW * s = menu_sub (m);

    unpost_menu (m);    /* unpost menu */
    werase (w);    /* erase menu window */
    wrefresh (w);    /* refresh screen */
    delwin (s);    /* delete menu windows */
    delwin (w);
}
```

Function **keypad()** is called with a second argument of 1 to enable virtual keys **KEY_LL**, **KEY_LEFT**, and others to be properly interpreted in the routine **get_request()** described in "Menu Driver Processing." See the discussion of **keypad()** in the **curses(S)** manual page for details. Finally, note the placement of checks for error returns in this example.

Menu Driver Processing

The `menu_driver()` is the workhorse of the menu system. Once the menu is posted, the `menu_driver()` handles all interaction with the end user. It responds to the following:

- item navigation requests
- menu scrolling requests
- item selection requests
- pattern buffer requests.

SYNTAX

```
int menu_driver (menu, c)
MENU * menu;
int c;
```

Your application program passes a character to the menu driver for processing and evaluates the results.

To enable your application program to fetch the character for the menu driver, write a routine that defines the input key virtualization. This is the correspondence between specific input keys, control characters, or escape sequences on the one hand and menu driver requests on the other. The virtualization routine returns a specific menu request or application command that the menu driver can process. Upon return from the menu driver, your application can check if the input was processed appropriately. If not, your application specifies the action to be taken. These actions may include terminating interaction with the menu, responding to help requests, generating an error message, and so forth.

Defining the Key Virtualization Correspondence

To illustrate a key virtualization routine, consider Figure 10-22, which shows the key virtualization routine `get_request()`. Nearly all the values it returns are the ETI menu requests to be discussed in the following sections.

Figure 10-22 Sample Routine that Translates Keys into Menu Requests

```

/* define application commands */

#define QUIT      (MAX_COMMAND + 1)

/* Note that ^X represents the character control-X.

^Q          - end menu processing

^N          - move to next item
^P          - move to previous item
home key    - move to first item
home down   - move to last item

left arrow  - move left to item
right arrow - move right to item
down arrow  - move down to item
up arrow    - move up to item

^U          - scroll up a line
^D          - scroll down a line
^B          - scroll up a page
^F          - scroll down a page

^X          - clear pattern buffer
^H <BS>    - delete character from pattern buffer
^A          - request next pattern match
^Z          - request previous pattern match

^T          - toggle item      */

static int get_request (w)      /* virtual key mapping */
WINDOW * w;
{
    int c = wgetch (w);        /* read a character */

    switch (c)
    {
        case 0x11:             /* ^Q */      return    QUIT;

        case 0x0e:             /* ^N */      return    REQ_NEXT_ITEM;
        case 0x10:             /* ^P */      return    REQ_PREV_ITEM;
        case KEY_HOME:         return    REQ_FIRST_ITEM;
        case KEY_LL:           return    REQ_LAST_ITEM;

        case KEY_LEFT:         return    REQ_LEFT_ITEM;
        case KEY_RIGHT:        return    REQ_RIGHT_ITEM;
        case KEY_UP:           return    REQ_UP_ITEM;
        case KEY_DOWN:         return    REQ_DOWN_ITEM;
    }
}

```

(Continued on next page.)

Menu Driver Processing

Figure 10-22 Sample Routine that Translates Keys into Menu Requests
(Continued)

```
case 0x15: /* ^U */ return REQ_SCR_ULINE;
case 0x04: /* ^D */ return REQ_SCR_DLINE;
case 0x06: /* ^F */ return REQ_SCR_DPAGE;
case 0x02: /* ^B */ return REQ_SCR_UPAGE;

case 0x18: /* ^X */ return REQ_CLEAR_PATTERN;
case 0x08: /* ^H */ return REQ_BACK_PATTERN;
case 0x01: /* ^A */ return REQ_NEXT_MATCH;
case 0x1a: /* ^Z */ return REQ_PREV_MATCH;

case 0x14: /* ^T */ return REQ_TOGGLE_ITEM;
}
return c;
}
```

Note that because **wgetch()** here automatically does a refresh before reading a character, you can omit explicit calls to **wrefresh()** in applications that do character input.

ETI Menu Requests

ETI menu requests are made by calling function **menu_driver()** with an **int** value that signifies the request. To appreciate the effects of some requests, bear in mind what a menu page is.

A menu page is the collection of currently visible menu items, i.e., those displayed in the menu subwindow.

A menu page is distinct from a form page, which is a logical portion of a form. Form pages are treated in the upcoming section “Forms.”

Item Navigation Requests

These requests enable your end user to navigate from item to item whether or not the items are displayed at the moment.

REQ_NEXT_ITEM	- move to next item
REQ_PREV_ITEM	- move to previous item
REQ_FIRST_ITEM	- move to first item
REQ_LAST_ITEM	- move to last item

The order of the items in the array originally passed to `new_menu()` or `set_menu_items()` determines the order in which items are visited in response to these requests.

A `REQ_NEXT_ITEM` request from the last item or a `REQ_PREV_ITEM` request from the first item returns the value `E_REQUEST_DENIED`.

Often, a scrolling operation not explicitly requested by the user may nonetheless take place in response to these requests. For example, the `REQ_FIRST_ITEM` request on a menu that is not currently displaying the first item may scroll to display the menu's first item at the top of the screen.

Directional Item Navigation Requests

These requests enable your end user to navigate from item to item in different directions.

<code>REQ_LEFT_ITEM</code>	- move left to item
<code>REQ_RIGHT_ITEM</code>	- move right to item
<code>REQ_UP_ITEM</code>	- move up to item
<code>REQ_DOWN_ITEM</code>	- move down to item

Directional item navigation requests are not cyclic. If there is no item on the current page to the left or right of the current item, the menu driver returns `E_REQUEST_DENIED` in response to the corresponding request.

On the other hand, if the menu is scrollable and there are more items above or below the current menu page, the corresponding requests `REQ_UP_ITEM` and `REQ_DOWN_ITEM` generate an automatic scrolling operation. If not, the menu driver returns `E_REQUEST_DENIED`.

Menu Scrolling Requests

These requests enable your users to scroll easily through menus that span more than one menu page.

<code>REQ_SCR_DLINE</code>	- scroll menu down a line
<code>REQ_SCR_ULINE</code>	- scroll menu up a line
<code>REQ_SCR_DPAGE</code>	- scroll menu down a page
<code>REQ_SCR_UPAGE</code>	- scroll menu up a page

The current and top items are adjusted by these operations.

Menu Driver Processing

Menu scrolling requests are also not cyclic. Attempts to scroll up from the first menu page, or scroll down from the last, return from the menu driver the value `E_REQUEST_DENIED`.

Multi-Valued Menu Selection Request

This request enables your end user to select or deselect an item in a multi-valued menu.

`REQ_TOGGLE_ITEM` - select/deselect item

If the item is currently selected, this request deselects it, and vice versa.

To use this request, the `O_ONEVALUE` option must be off. (See “Setting and Fetching Menu Options.”) If the option is on, you have a single-valued menu. In that case, this request fails and `E_REQUEST_DENIED` is returned from the menu driver.

Pattern Buffer Requests

The pattern buffer is an area automatically allocated for your menu application programs. It is used to position the current menu item at an item name that matches the pattern. You can modify the pattern buffer

- by calling `set_menu_pattern()` (described below)
- by passing the menu driver printable ASCII characters one at a time

Each nonprintable ASCII character that is received by the menu driver is assumed to be a menu request. On the other hand, each printable ASCII character that is received by the menu driver is entered into the pattern buffer. At the same time, the current item advances to the first matching item. If no matching item is found, the current item remains unchanged, the character is deleted from the pattern buffer, and the menu driver returns `E_NO_MATCH`.

The following requests enable you to change and read the pattern buffer.

<code>REQ_CLEAR_PATTERN</code>	- clear pattern buffer
<code>REQ_BACK_PATTERN</code>	- delete last character from pattern buffer
<code>REQ_NEXT_MATCH</code>	- move to next pattern match
<code>REQ_PREV_MATCH</code>	- move to previous pattern match

Request `REQ_CLEAR_PATTERN` clears the pattern buffer entirely.

Note

Without request `REQ_CLEAR_PATTERN`, the pattern buffer is automatically cleared after each successful scrolling or item navigation operation. In other words, anytime the top item or current item changes, the pattern buffer is cleared automatically.

`REQ_BACK_PATTERN` deletes the last character from the pattern buffer. This request can be used to support a backspace operation on the pattern buffer.

Sometimes more than one menu item will match the character(s) entered by the user. `REQ_NEXT_MATCH` moves the user forward on the displayed menu to the next array item that matches the data in the pattern buffer. `REQ_PREV_MATCH`, on the other hand, moves the user backward on the displayed menu to the previous array item that matches the pattern buffer. In both cases, if no additional match is found, the current item remains unchanged and `E_NO_MATCH` is returned from the menu driver.

Requests `REQ_NEXT_MATCH` and `REQ_PREV_MATCH` are cyclic through all menu items. In addition, these requests generate automatic scrolling requests if the menu is scrollable and the next or previous matching item is not visible.

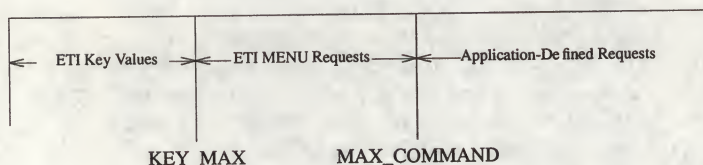
Note

An empty pattern buffer matches all items.

Application-Defined Commands

ETI menu requests are implemented as integers above the **curses** maximum key value `KEY_MAX`. A symbolic constant `MAX_COMMAND` is provided to enable your applications to implement their own requests (commands) without conflicting with the ETI form and menu system. All menu requests are greater than `KEY_MAX` and less than or equal to `MAX_COMMAND`. Your application-defined requests should be greater than `MAX_COMMAND`. Two illustrations are given in the following example. Figure 10-23 diagrams this relationship between ETI key values, ETI menu requests, and your application program's menu requests.

Figure 10-23 Integer Ranges for ETI Key Values and MENU Requests



Calling the Menu Driver

The menu driver checks whether the virtualized character passed to it is an ETI menu request. If so, it performs the request and reports the results. If the character is not a menu request, the menu driver checks if the character is data, i.e., a printable ASCII character. If so, it enters the character in the pattern buffer and looks for the first match among the item names. If no match is found, the menu driver deletes the character from the pattern buffer and returns `E_NO_MATCH`. If the character is not recognized as a menu request or data, the menu driver assumes the character is an application-defined command and returns `E_UNKNOWN_COMMAND`.

To illustrate a sample design for calling the menu driver, we will consider a program that permits interaction with a menu of astrological signs. Figure 10-24 displays the menu.

Figure 10-24 Sample Menu Output (S)

Aries	The Ram	
Taurus	The Bull	
Gemini	The Twins	
Cancer	The Crab	
Leo	The Lion	
Virgo	The Virgin	
Libra	The Balance	
Scorpio	The Scorpion	
Sagittarius	The Archer	
Capricorn	The Goat	
Aquarius	The Water Bearer	
Pisces	The Fishes	

You have already seen much of the astrological sign program in previous examples. Its function **get_request()**, for instance, appeared in Figure 10-22. Figure 10-25 shows its remaining routines.

Menu Driver Processing

Figure 10-25 Sample Program Calling the Menu Driver

```
/* This program displays a sample menu.

   Omitted here are the key mapping defined by get_request()
   in Figure 10-22; application-defined routines
   display_menu() and erase_menu() in Figure 10-21; and the
   curses initialization routine start_curses in section,
   "ETI Low-Level Interface to High-Level Functions" */

#include <string.h>
#include <menu.h>

static char *   PGM   = (char *) 0;   /* program name */

static int my_driver (m, c)   /* handle application commands */
MENU * m;
int c;
{
    switch (c)
    {
        case QUIT:
            return TRUE;
            break;
    }
    beep ();   /* signal error */
    return FALSE;
}

main (argc, argv)
int argc;
char * argv[];
{
    WINDOW *   w;
    MENU *     m;
    ITEM **    i;
    ITEM **    make_items ();
    void       free_items ();
    int        c, done = FALSE;

    PGM = argv[0];
    start_curses ();

    if (! (m = new_menu (make_items ())))
        error ("error return from new_menu", NULL);

    display_menu (m);
}
```

(Continued on next page.)

Figure 10-25 Sample Program Calling the Menu Driver

(Continued)

```

/* interact with user */
w = menu_win (m);
while (! done)
{
    switch (menu_driver (m, c = get_request (w)))
    {
        case E_OK:
            break;
        case E_UNKNOWN_COMMAND:
            done = my_driver (m, c);
            break;
        default:
            beep ();    /* signal error */
            break;
    }
}
erase_menu (m);
end_curses ();
i = menu_items (m);
free_menu (m);
free_items (i);
exit (0);
}

typedef struct
{
    char *    name;
    char *    desc;
}

ITEM_RECORD;

/* item definitions */

static ITEM_RECORD signs [] =
{
    "Aries",      "The Ram",
    "Taurus",     "The Bull",
    "Gemini",     "The Twins",
    "Cancer",     "The Crab",
    "Leo",        "The Lion",
    "Virgo",      "The Virgin",
    "Libra",      "The Balance",
    "Scorpio",    "The Scorpion",
    "Sagittarius", "The Archer",
    "Capricorn",  "The Goat",
    "Aquarius",   "The Water Bearer",
    "Pisces",     "The Fishes",
    (char *) 0,   (char *) 0,
};

```

(Continued on next page.)

Menu Driver Processing

Figure 10-25 Sample Program Calling the Menu Driver

(Continued)

```
#define MAX_ITEM    512

static ITEM *      items [MAX_ITEM + 1]; /* item buffer */

static ITEM ** make_items () /* create the items */
{
    int i;

    for (i = 0; i < MAX_ITEM && signs[i].name; ++i)
        items[i] = new_item (signs[i].name, signs[i].desc);

    items[i] = (ITEM *) 0;
    return items;
}

static void free_items (i) /* free the items */
ITEM ** i;
{
    while (*i)
        free_item (*i++);
}
```

Function **main()** first calls the application-defined routine **make_items()** to create the items from the array **signs**. The value returned is passed to **new_menu()** to create the menu. Function **main()** then initializes **curses** using **start_curses()** and displays the menu using **display_menu()**.

In its **while** loop, **main()** repeatedly calls **menu_driver()** with the character returned by **get_request()**. If the menu driver does not recognize the character as a request or data, it returns **E_UNKNOWN_COMMAND**, whereupon the application-defined routine **my_driver()** is called with the same character. Routine **my_driver()** processes the application-defined commands. In this example, there is only one, **QUIT**. If the character passed does not signify **QUIT**, **my_driver()** signals an error and returns **FALSE**—the signal prompts the user to re-enter the character. If the character passed is the **QUIT** character, **my_driver()** returns **TRUE**. In turn, this sets **done** to **TRUE**, and the **while** loop is exited.

Finally, **main()** erases the menu, terminates low-level ETI (**curses**), frees the menu and its items, and exits the program.

This example shows a typical design for calling the menu driver, but it is only one of several ways you can structure a menu application. For another example, see the demonstration program **menu2.c** delivered with the ETI product.

If the `menu_driver()` recognizes and processes the input character argument, it returns `E_OK`. In the following error situations, the `menu_driver()` returns the indicated value:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu
<code>E_BAD_STATE</code>	- called from init/term routines
<code>E_NOT_POSTED</code>	- menu is not posted
<code>E_UNKNOWN_COMMAND</code>	- unknown command
<code>E_NO_MATCH</code>	- item match failed
<code>E_REQUEST_DENIED</code>	- recognized request failed

Note

Because the menu driver calls the initialization and termination routines described in the next section, it may not be called from within them. Any attempt to do so returns `E_BAD_STATE`.

Establishing Item and Menu Initialization and Termination Routines

Sometimes, you may want the menu driver to execute a specific routine during the change of an item or menu. The following functions let you do this easily.

SYNTAX

```
typedef void (*PTF_void) ();

int set_menu_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void menu_init (menu)
MENU * menu;

int set_menu_term (menu, func)
MENU * menu;
PTF_void func;

PTF_void menu_term (menu)
MENU * menu;

int set_item_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_init (menu)
MENU * menu;

int set_item_term (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_term (menu)
MENU * menu;
```

The argument **func** is a pointer to the specific function you want executed by the menu driver. This application-defined function takes a menu pointer as an argument.

If you want your application to execute an application-defined function at one of the initialization or termination points listed below, you should call the appropriate **set_** routine at the start of your program. If you do not want a specific function executed in these cases, you may refrain from calling these routines altogether.

The following paragraphs summarize when each initialization and termination routine is executed.

Function `set_menu_init()`

The argument **func** to this function is automatically called by the menu system:

- Just before the menu is posted
- just after each menu scrolling operation, i.e., every time the top row changes on a posted menu, whether by the menu driver in response to a request or by a program's call to `set_current_item()` or `top_row()`

Function `set_item_init()`

The argument **func** is automatically called by the menu system:

- just before the menu is posted
- just after the current item on a posted menu is changed, whether by the menu driver's response to a request or by a program's call to `set_current_item()` or `top_row()`

Function `set_item_term()`

The argument **func** is automatically called by the menu system:

- just before the current item changes on a posted menu
- just before the menu is unposted

Function `set_menu_term()`

The argument **func** is automatically called by the menu system:

- just before a scrolling operation on a posted menu
- just before the menu is unposted

Menu Driver Processing

If functions `set_menu_init()`, `set_menu_term()`, `set_item_init()`, or `set_item_term()` encounter an error, they return

`E_SYSTEM_ERROR` - system error

Figure 10-26 shows how you can use function `set_item_init()` to implement a menu prompting feature as your end user moves from item to item.

Figure 10-26 Using an Initialization Routine to Generate Item Prompts

```
WINDOW * prompt_window;

void display_prompt (s)
char * s;
{
    WINDOW * w = prompt_window;

    werase (w);
    wmove (w, 0, 0);      /* move to window origin */
    waddstr (w, s);       /* write prompt in window */
    wrefresh (w);         /* display prompt */
}

void generate_prompt (m)
MENU * m;
{
    /* display the prompt string associated
       with the current item */

    char * s = item_userptr (current_item (m));
    display_prompt (s);
}

ITEM * items[NUMBER_OF_ITEMS + 1];

main ()
{
    MENU * m;

    for (i = 0; i < NUMBER_OF_ITEMS; ++i)
    {
        /* read in name and prompt strings here */

        items[i] = new_item (name, "");
        set_item_userptr (items[i], prompt);
    }
    items[i] = (ITEM *) 0;

    m = new_menu (items);
    set_item_init (m, generate_prompt); /* set initialization
                                         routine */
}
```


Function **set_item_init()** arranges to call **generate_prompt()** whenever the menu item changes. Function **generate_prompt()** fetches the item user pointer associated with the current item and calls **display_prompt()**, which displays the item prompt. Function **display_prompt()** is a separate function to enable you to use it for other prompts as well.

Fetching and Changing the Current Item

The current item is the item where your end user is positioned on the screen. Unless it is invisible, this item is highlighted and the cursor rests on the item. To have your application program set or determine the current item, use the following functions.

SYNTAX

```
int set_current_item (menu, item)
MENU * menu;
ITEM * item;

ITEM * current_item (menu)
MENU * menu;

int item_index (item)
ITEM * item;
```

Function **set_current_item()** enables you to set the current item by passing an item pointer, while function **current_item()** returns the pointer to the current item.

The function **item_index()** takes an item pointer argument and returns the index to that item in the item pointer array. The value of this index ranges from 0 through N-1, where N is the total number of items connected to the menu.

Because the menu driver satisfies ETI-defined item navigation requests automatically, your application program need not call **set_current_item()**, unless you want to implement additional item navigation requests for your application. You may, for instance, want a request to jump to a particular item or an item, say, two items down from the current one on the menu page.

When a menu is created by **new_menu()** or the items associated with a menu are changed by **set_menu_items**, the current item is set to the first item of the menu.

Menu Driver Processing

As an example of `set_current_item()`, the following function sets the current item of menu `m` to the first item of the menu:

```
int set_first_item (m) /* set current item to first item */
MENU * m;
{
    ITEM ** i = menu_items (m);
    return set_current_item (m, i[0]);
}
```

As an example of `current_item()`, the following routine checks if the first menu item is the current one:

```
int first_item (m) /* check if current item is first item */
MENU * m;
{
    ITEM * i = current_item (m);
    return item_index (i) == 0;
}
```

If successful, function `set_current_item()` returns `E_OK`. If an error occurs, function `set_current_item()` returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu pointer or item not connected to menu
<code>E_BAD_STATE</code>	- called from initialization or termination routines

Function `current_item()` returns `(ITEM *) 0` if given a `NULL` menu pointer or there are no items connected to the menu.

Function `item_index()` returns `-1` if the item pointer is `NULL` or the item is not connected to a menu.

Fetching and Changing the Top Row

Function **top_row()** returns the number of the menu row currently displayed at the top of your end user's menu. Function **set_top_row()** sets the top of the menu to the named row, unless the row does not start a complete page of items. In this case, it returns **E_BAD_ARGUMENT**.

```
int set_top_row(menu, row)
MENU * menu;
int row;
```

```
int top_row(menu)
MENU * menu;
```

Function **set_top_row()** sets the current item to the leftmost item in the new top row. Variable **row** must be in the range of 0 through **TR-VR**, where **TR** is the total number of rows as determined by the menu format and **VR** is the number of visible rows. If the value of **row** is greater, the row does not start a complete page of items. See "Specifying the Menu Format" for details on menu display.

When a menu is created by **new_menu()** or the items associated with the menu are changed by **set_menu_items**, the top row is set to 0.

Note

If the menu format or the **O_ROWMAJOR** option is changed, the top row is automatically set to 0. See "Specifying the Menu Format" and "Setting and Fetching Menu Options" for details on changing these menu attributes.

In addition, if the current item is changed by **set_current_item()** or **set_menu_pattern()** to an item that is not currently visible, the top row is generally set to the row that contains the new current item. The sole exception occurs when, as noted above, the top row does not start a complete page of items.

If successful, function **set_top_row()** returns **E_OK**. If an error occurs, **set_top_item()** returns one of the following error messages:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- NULL menu pointer or index out of range
E_BAD_STATE	- called from init/term routines
E_NOT_CONNECTED	- no connected items

Function **top_row()** returns -1 if given a NULL menu pointer or no items are connected to the menu.

Positioning the Menu Cursor

Some applications may need to move the menu's window cursor from the position required for continued processing by the ETI menu driver. To move the cursor back to where it belongs, you use function `pos_menu_cursor()`.

SYNTAX

```
int pos_menu_cursor (menu)
MENU * menu;
```

If your application does not change the cursor position in the menu window, it will not be necessary to call this function.

Your application might change the cursor position automatically as the result of prior calls to menu driver initialization routines such as `set_item_init()`. Or it might do so as the result of explicit calls to application routines such as writing a prompt. Figure 10-27 illustrates this usage.

Figure 10-27 Returning Cursor to its Correct Position
for Menu Driver Processing

```
void generate_prompt (m)
MENU * m;
{
    /* display the prompt string associated with
       the current item */

    WINDOW * w = menu_win (m);
    char * s = item_userptr (current_item (m));
    box (w, 0, 0);
    wmove (w, 0, 0);
    waddstr (w, s);
    pos_menu_cursor (m);
}
```

If function `pos_menu_cursor()` is successful, it returns `E_OK`. In the following error situations, it fails and returns the indicated value:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu pointer
<code>E_NOT_POSTED</code>	- menu is not posted

Changing and Fetching the Pattern Buffer

Remember that the pattern buffer is used to make the first item that matches the pattern the current item. In general, to match the current menu item, your application program inserts characters into the pattern buffer that have been passed to the menu driver from the user's data entry. As an alternative, you can insert characters into the pattern buffer with the function `set_menu_pattern()`.

SYNTAX

```
int set_menu_pattern (menu, pattern)
MENU * menu;
char * pattern;

char * menu_pattern (menu)
MENU * menu;
```

Function `set_menu_pattern()` first clears the pattern buffer and then adds the characters in **pattern** to the buffer until **pattern** is exhausted. The function next tries to find the first item that matches the **pattern**. If it does not find a complete match, the pattern buffer is cleared and the current item does not change. If **pattern** is the null string (""), the pattern buffer is simply cleared. The pattern buffer is automatically cleared in the following situations:

- Each successful scrolling or item navigation operation is completed (in other words, whenever the top or current item changes).
- A menu is created by `new_menu()`.
- The items associated with a menu are changed by `set_menu_items`.

If successful, function `set_menu_pattern()` returns `E_OK`. If an error occurs, function `set_menu_pattern()` returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- NULL menu pointer or NULL pattern pointer
<code>E_NO_MATCH</code>	- complete match failed

Function `menu_pattern()` returns the value of the string in the pattern buffer. If the pattern buffer is empty (the null string ""), it returns the null string (""). If the menu pointer argument is NULL, it returns NULL, i.e., (char *) 0.

Menu Driver Processing

To determine if your user has entered data that matches an item, you might write a routine that uses **set_menu_pattern()**, as follows.

```
int find_match (m, newpattern) /* returns TRUE or FALSE */
MENU * m;
char * newpattern;
{
    return set_menu_pattern(m, newpattern) == E_OK;
}
```

If the **newpattern** matches a menu item, function **set_menu_pattern()** returns **E_OK** and hence **find_match()** returns **TRUE**. In addition, **find_match()** advances the current item to the matching item.

Manipulating the Menu User Pointer

As it does for panels and forms, ETI provides user pointers for each menu. You can use these pointers to reference menu messages, titles, and the like.

SYNTAX

```
int set_menu_userptr (menu, userptr)
MENU * menu;
char * userptr;

char * menu_userptr (menu)
MENU * menu;
```

By default, the menu user pointer (what **menu_userptr()** returns) is **NULL**.

If successful, **set_menu_userptr()** returns **E_OK**. If an error occurs, it returns the following:

E_SYSTEM_ERROR - system error

The code in Figure 10-28 illustrates how you can use these two functions to display a title for your menu. Function **main()** sets the menu user pointer to point to the title of the menu. Later, function **display_menu()** initializes the title with the value returned by **menu_userptr()**. We have previously seen a version of **display_menu()** in Figure 10-25.

Manipulating the Menu User Pointer

Figure 10-28 Example Setting and Using A Menu User Pointer

```
static void display_menu (m)      /* create menu windows and post */
MENU * m;
{
    char *      title = menu_userptr (m);  /* fetch menu title */
    WINDOW *    w;
    int         rows;
    int         cols;

    scale_menu (m, &rows, &cols);  /* get dimensions of menu */
    /* create menu window and subwindow */
    if (w = newwin (rows+2, cols+2, 0, 0))
    {
        set_menu_win (m, w);
        set_menu_sub (m, derwin (w, rows, cols, 1, 1));
        box (w, 0, 0);
        keypad (w, 1);
    }
    else
        error ("error return from newwin", NULL);

    if (post_menu (m) != E_OK)
        error ("error return from post_menu", NULL);
    if (title)
        /* if title set */
    {
        size = strlen (title);
        wmove (w, 0, (cols-size)/2+1); /* position cursor */
        waddstr (w, title);             /* write title */
    }
}

main ()
{
    MENU * m;
    char * menutitle; /* initialize menutitle to desired string */

    set_menu_userptr (m, menutitle); /* set user pointer to point to title */
    display_menu (m);
}
```

If function **set_menu_userptr()** is passed a NULL menu pointer, like all ETI functions, it assigns a new current default menu user pointer. In the following, the new default is the string "Default Menu Title."

```
MENU * m;

char * userptr = "Default Menu Title";

set_menu_userptr ( (MENU *) 0, userptr); /* sets new default userptr */
```

Setting and Fetching Menu Options

ETI provides several menu options, some of which we have already discussed. Two functions manipulate options: one sets them, the other returns their settings.

SYNTAX

```
int set_menu_opts (menu, opts)
MENU * menu;
OPTIONS opts;

OPTIONS menu_opts (menu)
MENU * menu;

options:
    O_ONEVALUE
    O_SHOWDESC
    O_ROWMAJOR
    O_IGNORECASE
    O_SHOWMATCH
    O_NONCYCLIC
```

Besides turning the named options on, function **set_menu_opts()** turns off all other menu options. By default, all menu options are on.

The menu options and their effects are as follows:

O_ONEVALUE	determines whether the menu is a single-valued or multi-valued. In general, menus are single-valued and this option is on. Recall that upon exit from single-valued menus, your application queries the current item to ascertain the item selected. Turning off this option signifies a multi-valued menu. One way to select several items is to use the REQ_TOGGLE_ITEM request, another is to call set_item_value() . (See the previous section “Manipulating an Item’s Select Value in a Multi-Valued Menu.”) Recall that your application must examine each item’s select value to determine whether it has been selected. When this option is on, all item select values are FALSE .
-------------------	---

Setting and Fetching Menu Options

- O_SHOWDESC** determines whether or not the description of an item is displayed. By default, this option is on and both the item name and description are displayed. If this option is off, only the name is displayed.
- O_ROWMAJOR** determines how the menu items are presented on the screen—in row-major or column-major order. In row-major order, menu items are first displayed left to right, then top to bottom. In column-major order, they are displayed first top to bottom, then left to right. By default, this option is on, so menu items are displayed in row-major order. If the option is off, the items are displayed in column-major order. See “Specifying the Menu Format” for more details on how menus are displayed.
- O_IGNORECASE** instructs the menu driver to ignore upper- and lower-case during the item match operation. If this option is off, character case is not ignored and the match must be exact.
- O_SHOWMATCH** determines whether visual feedback is provided as each item’s data entry is processed. Ordinarily, as soon as a match occurs, the cursor is advanced through the item to reflect the contents of the pattern buffer. If this option is off, however, the cursor remains to the left of the current item.
- O_NONCYCLIC** determines how `REQ_NEXT_ITEM` and `REQ_PREV_ITEM` behave when the current item is the last or first item. Default is to not allow cycling from first to last item or from last to first item. If `O_NONCYCLIC` is turned off, `REQ_NEXT_ITEM` from the last item causes the first item to become the current. If `O_NONCYCLIC` is turned off, `REQ_PREV_ITEM` from the first item causes the last item to become current.

Setting and Fetching Menu Options

Like all ETI options, menu **OPTIONS** are Boolean values, so you use Boolean operators to turn them on or off with functions **set_menu_opts()** and **menu_opts()**. For example, to turn off option **O_SHOWDESC** for menu **m0** and turn on the same option for menu **m1**, you can write:

```
MENU * m0, * m1;

set_menu_opts (m0, menu_opts (m0) & ~O_SHOWDESC); /* turn option off */
set_menu_opts (m1, menu_opts (m1) | O_SHOWDESC); /* turn option on */
```

ETI provides two alternative functions for turning options on and off for a given menu.

SYNTAX

```
int menu_opts_on (menu, opts)
MENU * menu;
OPTIONS opts;

int menu_opts_off (menu, opts)
MENU * menu;
OPTIONS opts;
```

Unlike function **set_menu_opts()**, these functions do not affect options that are unmentioned in their second argument. In addition, if you want to change one option, you need not apply Boolean operators or use **menu_opts()**.

For example, the following code turns option **O_SHOWDESC** off for menu **m0** and on for menu **m1**:

```
MENU * m0, * m1;

menu_opts_off (m0, O_SHOWDESC); /* turn option off */
menu_opts_on (m1, O_SHOWDESC); /* turn option on */
```

As usual, you can change the current default for each option by passing a **NULL** menu pointer. For instance, to turn the default option **O_SHOWDESC** off, you write

```
menu_opts_off ((MENU *) 0, O_SHOWDESC); /* turn default option off */
```

In general, functions **set_menu_opts()**, **menu_opts_on()**, and **menu_opts_off()** return **E_OK**. If an error occurs, they return one of the following:

E_SYSTEM_ERROR	- system error
E_POSTED	- menu is posted

Forms

A form is a collection of one or more pages of fields. The fields may be used for titles, labels to guide the user, or for data entry. Figure 10-29 displays a simple form with five fields including two for data entry.

Figure 10-29 Sample Form Display

Sample Form

Field 1: _____
Field 2: _____

Compiling and Linking Form Programs

To use the form routines, you specify

```
#include <form.h>
```

in your C program files and compile and link with the command line

```
cc [ flags ] files -lform -lcurses [ libraries ]
```

If you want to use the menu or panel routines as well, place the appropriate **-l** option before the option **-lcurses**.

Overview: Writing Form Programs in ETI

This section introduces the basic ETI form terminology, lists the steps in a typical form application, and reviews the sample program that produced the output as shown in Figure 10-29.

Some Important Form Terminology

The following terms are helpful in working with ETI form functions:

field	an $m \times n$ block of character positions within a form that ETI functions can manipulate as a unit
active field	a field that is visited during form processing for data entry, change, selection, and so forth
inactive field	a field that is completely ignored during form processing, such as a title, field marker or other label
form	a collection of one or more pages of fields
connecting fields to a form	associating an array of field pointers with a form
page	a logical subdivision of a form usually occupying one screen
posting a form	writing a form on its associated subwindow
unposting a form	erasing a form from its associated subwindow

freeing a form	deallocating the memory for a form and, as a by-product, disconnecting the previously associated array of field pointers from the form
freeing a field	deallocating the memory for a field
NULL	generic term for a null pointer cast to the type of the particular object—field, form, and so on

What a Typical Form Application Program Does

In general, a form application program will

- initialize low-level ETI (**curses**)
- create the fields for the form
- create the form
- post the form
- refresh the screen
- process end user form requests
- unpost the form
- free the form
- free the fields
- terminate low-level ETI (**curses**)

A Sample Form Application Program

Figure 10-30 shows the ETI program necessary for producing the form in Figure 10-29.

Overview: Writing Form Programs in ETI

Figure 10-30 Code To Produce a Simple Form

```
#include <form.h>
#include <string.h>

FIELD * make_label (frow, fool, label)
int frow;          /* first row */
int fool;          /* first column */
char * label;      /* label */
{
    FIELD * f = new_field (1, strlen (label), frow, fool, 0, 0);

    if (f)
    {
        set_field_buffer (f, 0, label);
        set_field_opts (f, field_opts(f) & ~O_ACTIVE);
    }
    return f;
}

FIELD * make_field (frow, fool, cols)
int frow;          /* first row */
int fool;          /* first column */
int cols;          /* number of columns */
{
    FIELD * f = new_field (1, cols, frow, fool, 0, 0);

    if (f)
        set_field_back (f, A_UNDERLINE);
    return f;
}

main ()
{
    FORM *      form;
    FIELD *      f[6];
    int          i = 0;

    /*
     * ETI initialization
     */
    initscr ();
    nonl ();
    raw ();
    noecho ();
    wclear (stdscr);
}
```

(Continued on next page.)

Figure 10-30 Code To Produce a Simple Form

(Continued)

```

/*
  create fields
*/
f[0] = make_label (0, 7, "Sample Form");
f[1] = make_label (2, 0, "Field 1:");
f[2] = make_field (2, 9, 16);
f[3] = make_label (3, 0, "Field 2:");
f[4] = make_field (3, 9, 16);
f[5] = (FIELD *) 0;

/*
  create and display form
*/
form = new_form (f);
post_form (form);
wrefresh (stdscr);
sleep (5);

/*
  erase form and free both form and fields
*/
unpost_form (form);
wrefresh (stdscr);
free_form (form);

while (f[i])
    free_field (f[i++]);

/*
  ETI termination
*/
endwin ();
exit (0);
}

```

In this example, all text within the form is associated with a field. Fields may be active or inactive: active fields are affected by form processing, inactive fields are not. The underlined fields are active, whereas the label fields, "Sample Form", "Field 1:", and "Field 2:", are inactive.

Turn now to the program itself. This example starts with two **#include** files. Every form program must include the header file **form.h**, which contains important definitions of form objects. This particular program uses the C string library function **strlen()**, so it includes the header file **string.h**, whose definitions the string library function needs. See **string(S)** in the *Programmer's Reference* for details.

Overview: Writing Form Programs in ETI

Next, there are two programmer-defined functions **make_label()** and **make_field()**, which we will discuss in a moment. Consider procedure **main()**. It declares three objects:

- **form**, a pointer to a form
- **f[6]**, an array of field pointers
- **i**, an index variable, initialized to 0

The first five functions initialize low-level ETI (**curses**) for high-level ETI form functions. Function **initscr()** initializes the screen, **nonl()** ensures that a carriage return on using **wgetch()** will not automatically generate a newline, **raw()** passes input characters uninterpreted to your program, **noecho()** disables echoing of your user's input (the form functions provide echoing where appropriate), and **wclear(stdscr)** clears the standard screen.

The statements that create the form's fields and labels in this example make calls to the programmer-defined functions **make_label()** and **make_field()**. You can do without these programmer-defined functions, but you may find them convenient. Both of them use the ETI function **new_field()**. They take three arguments, which correspond to three of the six arguments of **new_field()**.

The first argument of **new_field()** is the number of rows of the field. In this example, it is always one. The last two arguments are often 0 as they are here; they will be explained in the next section. The second argument of **new_field()** is the number of columns in the field. This number is determined from the third parameter in **main()**'s calls to **make_label()** and **make_field()**. For the label fields, the calls to **make_label()** pass the string that is to constitute the field so that **strlen()** can be used to count the length or number of columns of the string. For the fields to be edited by the end-user (had this example permitted entering data into the fields), calls to **make_field()** simply pass the number of columns directly.

The third and fourth arguments to **new_field()** correspond to the first and second arguments to **make_label()** and **make_field()**. They are the starting position (**firstrow**, **firstcol**) of the label or field in the form subwindow. (In this example, the default subwindow **stdscr** is used.) The last assignment to **f[5]** terminates the array with the NULL field pointer.

Once the function **make_label()** creates the field for the label, it places the label in the field using function **set_field_buffer()**. The second argument to this function is 0 because the value of a field is stored in buffer 0. Finally, function **make_label()** calls **set_field_opts()**, which turns off the **O_ACTIVE** option for the field. This means that the field is ignored during form driver processing.

On the other hand, once the function **make_field()** creates the field proper, it sets the field's background attribute to **A_UNDERLINE**. This has the effect of underlining the field so that it is visible.

After you create the fields for a form, you create the form itself using **new_form()**. This function takes the pointer to the array of field pointers and connects the fields to the form. The pointer returned is stored in variable **form**—it will be passed to subsequent form manipulation routines. To display the form, function **post_form()** posts it on the default subwindow **stdscr**, while **wrefresh(stdscr)** actually displays this subwindow on the terminal screen. The display remains for 5 seconds, as determined by **sleep()**.

At this point, most forms would accept and process user input. To illustrate a very simple form, this program does not accept user input.

To erase the form, you first unpost it using **unpost_form()**. This erases it from the form subwindow. The call to **wrefresh()** actually erases the form from the display screen. Function **free_form()** disconnects the form from its array of field pointers **f**.

The **while()** loop, starting with the first field in the field pointer array, frees each field referenced in the array. The effect is to deallocate the space for each field.

We have met the last two lines of the program before. Function **endwin()** terminates low-level ETI, while **exit(0)** terminates the program.

There are many ETI form routines not listed in Figure 10-30. These enable you to tailor your form programs to suit local needs and preferences. The following sections explain how to use all ETI form routines. Each routine is illustrated with one or more code fragments. Many of these are drawn from two larger form application programs listed at the end of the chapter. By reviewing the code fragments, you will come to understand the larger programs.

Creating and Freeing Fields

To create a form, you must first create its fields. The following functions enable you to create fields and later free them.

SYNTAX

```
FIELD * new_field (rows, cols, firstrow, firstcol, nrow, nbuf)
int rows, cols, firstrow, firstcol, nrow, nbuf;
```

```
FIELD * dup_field (field, firstrow, firstcol)
FIELD * field;
int firstrow, firstcol;
```

```
FIELD * link_field (field, firstrow, firstcol)
FIELD * field;
int firstrow, firstcol;
```

```
int free_field (field)
FIELD * field;
```

Unlike menu items which always occupy one row, the fields on a form may contain one or more rows. Function `new_field()` creates and initializes a new field that is **rows** by **cols** large and starts at point (**firstrow**, **firstcol**) relative to the origin of the form subwindow. All current system defaults are assigned to the new field when it is created using `new_field()`.

Variable **nrow** is the number of offscreen rows allocated for this field. Offscreen rows enable your program to display only part of a field at a given moment and let the user scroll through the rest. A zero value means that the entire field is always displayed, while a nonzero value means that the field is scrollable.

Variable **nbuf** is the number of additional buffers allocated for this field. You can use them to support default field values, undo operations, or other similar operations requiring one or more auxiliary field buffers.

Variables **rows** and **cols** must be greater than zero, while **firstrow**, **firstcol**, **nrow**, and **nbuf** must be greater than or equal to zero.

Each field buffer is $((\text{rows} + \text{nrow}) * \text{cols} + 1)$ characters large. (The extra character position holds the NULL terminator.) All fields have one buffer (namely, field buffer 0) that maintains the field's value. This buffer reflects any changes your end-user may make to the field. See the section "Setting and Reading Field Buffers" for more details.

To create a form field **occupation** one row high and 32 columns wide, starting at position **2,15** in the form subwindow, with no offscreen rows and no additional buffers, you can write:

```
FIELD * occupation;
```

```
occupation = new_field (1, 32, 2, 15, 0, 0); /* create field */
```

Generally you create all the fields for a form at the same point in your program, as Figure 10-30 demonstrated.

The function **dup_field()** duplicates an existing field at the new location **firstrow, firstcol**. During initialization, **dup_field()** copies nearly all the attributes of its field argument as well as its size and buffering information. However, certain attributes, such as being the first field on a page or having the field status set, are not duplicated in the newly created field. See the later sections, “Creating and Freeing Forms” and “Setting and Reading the Field Status,” for details on these attributes.

Like **dup_field()**, function **link_field()** duplicates an existing field at a new location on the same form or another one. Unlike **dup_field()**, however, **link_field()** arranges that the two fields share the space allocated for the field buffers. All changes to the buffers of one field appear also in the buffers of the other. Besides enabling your user to enter data into two or more fields at once, this function is useful for propagating field values to later pages where only the first field is active (currently open to form processing). In this case, the inactive fields in effect become dynamic labels. See the section below, “Manipulating Field Options.”

Note

Linked fields share only the space allocated for the field buffers—the attribute values of either field may be changed without affecting the other.

Creating and Freeing Fields

Consider field **occupation** in the previous example. To duplicate it at location **3,15** and link it at location **4,15**, you write

```
FIELD * dup_occ, * link_occ;

dup_occ = dup_field (occupation, 3, 15);
link_occ = link_field (occupation, 4, 15);
```

Functions **new_field()**, **dup_field()**, and **link_field()** return a NULL pointer if there is no available memory for the **FIELD** structure or if they detect an invalid parameter.

Function **free_field()** frees all space allocated for the given field. Its argument is a pointer previously obtained from **new_field()**, **dup_field()**, or **link_field()**.

Note

To free a field, be sure that the field is not connected to a form.

As described in the section below, “Creating and Freeing Forms,” you can disconnect fields from forms by using functions **free_form()** or **set_form_fields()**.

To free a form and all its fields, you write

```
FORM * form;

/* get pointer to form's field pointer array using
   form_fields() described in section below,
   "Changing the Fields on an Existing Form" */

FIELD ** f = form_fields (form);

free_form (form);          /* free form */

while (*f)
    free_field (*f++); /* free each field and increment pointer */
```

Notice that you free the form before its fields.

Creating and Freeing Fields

If successful, function **free_field()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field pointer
E_CONNECTED	- connected field

Remember that the field pointer returned by **new_field()**, **dup_field()**, or **link_field()** is passed to all field routines that record or examine the field's attributes. As with menu items, once a form field is freed, it must not be used again. Because the freed field pointer does not point to a genuine field, undefined results occur.

Manipulating Field Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A field attribute is a feature of a field whose value can be set or read by an appropriate ETI function. Field attributes include the field size and location.

Obtaining Field Size and Location Information

This function enables you to determine the defining characteristics of a field—its size, position, number of offscreen rows, and number of associated buffers.

SYNTAX

```
int field_info (field, rows, cols, firstrow, firstcol, nrow, nbuf)
FIELD * field;
int * rows, * cols, * firstrow, * firstcol, * nrow, * nbuf;
```

Because function **field_info()** must return more than a single value and C passes arguments to functions by “call by value” only, **field_info()** uses the pointer arguments **rows**, **cols**, **firstrow**, **firstcol**, **nrow**, and **nbuf**. These arguments are pointers to the locations used to return the requested information: the number of rows and columns comprising the field, the field starting location relative to the origin of its form subwindow, the number of offscreen rows, and the number of additional buffers.

As an example, consider how you might use **field_info()** to determine a field’s buffer size. You fetch the field’s number of onscreen and offscreen rows and number of columns, and do the arithmetic, thus:

```
int bufsize (f)
FIELD * f;
{
    int rows, cols, firstrow, firstcol, offrow, nbuf;
    field_info (f, &rows, &cols, &firstrow, &firstcol, &offrow, &nbuf);
    /* add up size of field and its terminator */
    return (rows + offrow) * cols + 1;
}
```

Note the use of the address operator **&** to pass **field_info()** the requisite pointers to the locations used to return the requested field information.

If successful, function `field_info()` returns `E_OK`. If not, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field pointer

Moving a Field

ETI provides the following function to move an existing disconnected field to a new location.

SYNTAX

```
int move_field (field, firstrow, firstcol)
FIELD * field;
int firstrow;
int firstcol;
```

Figure 10-31 shows one way you might use function `move_field()`. Function `shift_fields()` receives the `int` value `updown`, which it uses to change the row number of each field in a given field pointer array. You could, of course, shift the columns in like fashion.

Figure 10-31 Example Shifting All Form Fields a Given Number of Rows

```
void shift_fields (f, updown)
FIELD ** f;
int updown; /* signed number of rows to shift */
{
    int rows, cols, frow, fcol, nrow, nbuf;

    while (*f)
    {
        /* field_info() fetches the values of the field parameters */
        field_info (*f, &rows, &cols, &frow, &fcol, &nrow, &nbuf);
        move_field (*f, frow + updown, fcol);
        ++f;
    }
}
```

See the previous section, “Obtaining Field Size and Location Information,” for more on `field_info()` used in this example.

Manipulating Field Attributes

If successful, function **move_field()** returns **E_OK**. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field or firstrow/firstcol < 0
E_CONNECTED	- connected field

Changing the Current Default Values for Field Attributes

ETI establishes initial current default values for field attributes. During field initialization, every field attribute is assigned the current default value for the attribute. As you can with menu functions, you can change or retrieve the current default attribute values by calling the appropriate function with a **NULL** field pointer. After the current default changes, every field created using **new_field()** will have the new default value.

Note

Fields previously created do not have their attributes changed by changing the current system default.

Several of the following sections show how to change the default values for various field attributes.

Setting the Field Type To Ensure Validation

Every field is created with the current default field type. The initial ETI default field type is a **no_validation** field. Any data may occupy it. (This default can be changed as described below.) To change a field's type from the default, ETI provides the following functions for manipulating a field's (data) type.

SYNTAX

```
int set_field_type (field, type, [arg_1, arg_2, ...])
FIELD * field;
FIELDTYPE * type;

FIELDTYPE * field_type (field)
FIELD * field;

char * field_arg (field)
FIELD * field;
```

The function **set_field_type()** takes a **FIELDTYPE** pointer and a variable number of arguments depending on the field type. The field type ensures that the field is validated as your end-user enters characters into the field or attempts to leave it.

The form driver (described later in the section “Form Driver Processing”) validates the data in a field only when data is entered by your end-user. Validation does NOT occur when

- the application program changes the field value by calling **set_field_buffer()**
- linked field values are changed indirectly—by changing the field to which they are linked

In all cases, validation occurs only if data is changed by passing data or making requests to the form driver. To make requests, your user enters characters or escape sequences mapped to commands that the form driver recognizes. See the section below, “Form Driver Processing.”

If successful, **set_field_type()** returns **E_OK**. If not, it returns the following:

E_SYSTEM_ERROR	- system error
-----------------------	----------------

Function **field_type()** returns the field type of the field, while function **field_arg()** returns the field argument pointer. For more on the field argument pointer in programmer-defined field types, see the section below, “Supporting Programmer-Defined Field Types.”

Manipulating Field Attributes

If the function `set_field_type()` is applied to a NULL field, the field type becomes the new current default.

Note

Remember that the initial ETI default is not to validate the field at all; any kind of data may be entered into the field.

You can change the ETI default by giving function `set_field_type()` a NULL field pointer. Suppose, for instance, that you want to change the system default field type to a minimum 10-character field of type `TYPE_ALNUM`. As described below, this field type accepts alphanumeric data—every entered character must be a digit or an alphabetic (not a special) character. You can write

```
set_field_type ((FIELD *) 0, TYPE_ALNUM, 10);
```

ETI provides several generic field types besides `TYPE_ALNUM`. Moreover, you can define your own field types, as described later in the section “Creating and Manipulating Programmer-Defined Field Types.” The following sections describe all ETI generic field types.

TYPE_ALPHA

The form driver restricts a field of this type to alphabetic data.

SYNTAX

```
set_field_type (field, TYPE_ALPHA, width);  
int width; /* minimum token width */
```

`TYPE_ALPHA` takes one additional argument, the minimum width specification of the field. Note that when you previously create a field with function `new_field()`, your `cols` argument is the maximum width specification of the field. With `TYPE_ALPHA` (and `TYPE_ALNUM` as well), your specification **width** must be less than or equal to **cols**. If not, the form driver cannot validate the field.

Note

TYPE_ALPHA does not allow blanks or other special characters.

To set a middlename field, for instance, to TYPE_ALPHA with a minimum of 0 characters (in effect, to make the end-user's completing the field optional), you can write

```
FIELD * middlename;  
set_field_type (middlename, TYPE_ALPHA, 0);
```

TYPE_ALNUM

This type restricts the set field to alphanumeric data, alphabetic characters (upper- or lower-case) and digits.

SYNTAX

```
set_field_type (field, TYPE_ALNUM, width);  
int_width /* minimum token width */
```

Like TYPE_ALPHA, TYPE_ALNUM takes one additional argument, the field's minimum width specification.

Note

Like TYPE_ALPHA, TYPE_ALNUM does not allow blanks or other special characters.

To set a field, say **partnumber**, to receive alphanumeric data at least 8 characters wide, you write

```
FIELD * partnumber;  
  
set_field_type (partnumber, TYPE_ALNUM, 8);
```

Manipulating Field Attributes

TYPE_ENUM

This field type enables you to restrict the valid data for a field to a set of enumerated values. The type takes three arguments beyond the minimum two that `set_field_type()` requires.

SYNTAX

```
set_field_type (field, TYPE_ENUM, keyword_list, checkcase, checkunique);
char ** keyword_list;      /* list of acceptable values */
int checkcase;             /* check character case */
int checkunique;           /* check for unique match */
```

The argument **keyword_list** is a NULL-terminated array of pointers to character strings that are the acceptable enumeration values. Argument **checkcase** is a Boolean flag that indicates whether upper- or lower-case is significant during match operations. Finally, **checkunique** is a Boolean flag indicating whether a unique match is required. If it is off and your end-user enters only part of an acceptable value, the validation procedure completes the field value automatically with the first matching value in the type. If it is on, the validation procedure completes the field value automatically only when enough characters have been entered to make a unique match.

To create a field, say **response**, with valid responses of “yes” (“y”) or “no” (“n”) in upper- or lower-case, you write the following:

```
char * yesno[] = { "yes", "no", (char *)0 };
FIELD * response;

set_field_type (response, TYPE_ENUM, yesno, FALSE, FALSE);
```

For an example that sets the last field (**checkunique**) to TRUE, see Figure 10-32, which sets the TYPE_ENUM of field **color** to a list of colors.

Figure 10-32 Setting a Field to TYPE_ENUM of Colors

```
char * colors[13] =
{
    "Black",           "Charcoal",       "Light Gray",
    "Brown",           "Camel",         "Navy",
    "Light Blue",      "Hunter Green",  "Gold",
    "Burgundy",        "Rust",          "White",
    (char *) 0
};
FIELD * color;

set_field_type (color, TYPE_ENUM, colors, FALSE, TRUE);
```


Setting the field to TRUE requires the user to enter the seventh character of the color name in certain cases ("Light Blue" and "Light Gray") before a unique match is made.

TYPE_INTEGER

This type enables you to restrict the data in a field to integers.

SYNTAX

```
set_field_type (field, TYPE_INTEGER, precision, vmin, vmax);
int precision;      /* width for left padding with 0's */
long vmin;          /* minimum acceptable value */
long vmax;          /* maximum acceptable value */
```

TYPE_INTEGER takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity. A TYPE_INTEGER value is valid if it consists of an optional minus sign followed by some number of digits. As the end-user tries to leave the field, the range check is applied.

Note

If, contrary to possibility, the maximum value **vmax** is less than or equal to the minimum value **vmin**, the range check is ignored—any integer that fits in the field is valid.

If the range check is passed, the integer is padded on the left with zeros to the precision specification. For instance, if the current value were 18, a precision of 3 would display

018

whereas a precision of 4 would display

0018

For more on ETI's handling of precision, see the manual page **printf(S)** in the *Programmer's Reference*.

Manipulating Field Attributes

As an example of how to use `set_field_type()` with `TYPE_INTEGER`, the following might represent a month, padded to 2 digits:

```
FIELD * month;

set_field_type (month, TYPE_INTEGER, 2, 1L, 12L);
/* displays single digit months with leading 0 */
```

Note the requirement that the minimum and maximum values be converted to type **long** with the **L**.

TYPE_NUMERIC

This type restricts the data for the set field to decimal numbers.

SYNTAX

```
set_field_type (field, TYPE_NUMERIC, precision, vmin, vmax);
int precision; /* digits to right of the decimal point */
double vmin; /* minimum acceptable value */
double vmax; /* maximum acceptable value */
```

`TYPE_NUMERIC` takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity as decimal numbers. A `TYPE_NUMERIC` value is valid if it consists of an optional minus sign, some number of digits, a decimal point, and some additional digits.

The precision is not used in validation; it is used only in determining the output format. See **printf(S)** in the *Programmer's Reference* for more on precision. As the end-user tries to leave the field, the range check is applied.

As with `TYPE_INTEGER`, if the maximum value is less than or equal to the minimum value, the range check is ignored.

For instance, to set a maximum value of \$100.00 for a monetary field **amount**, you write:

```
FIELD * amount;

set_field_type (amount, TYPE_NUMERIC, 2, 0.00, 100.00);
```

TYPE_REGEX

This type enables you to determine whether the data entered into a field matches a specific regular expression.

SYNTAX

```
set_field_type (field, TYPE_REGEX, expression);  
char * expression; /* regular expression */
```

TYPE_REGEX takes one additional argument, the regular expression. See **regcmp(S)** in the *Programmer's Reference* or the chapter, "lex," in this Guide for regular expression details.

Consider, for example, how you might create a field that represents a part number with an upper- or lower-case letter followed by exactly 4 digits:

```
FIELD * partnumber;  
  
set_field_type (partnumber, TYPE_REGEX, "[A-Za-z][0-9]{4}$");
```

Note that this example assumes the field is 5 characters wide. If not, you may want to change the pattern to accept blanks on either side, thus:

```
FIELD * partnumber;  
  
set_field_type (partnumber, TYPE_REGEX, "^*[A-Za-z][0-9]{4}*$");
```

Justifying Data in a Field

Unlike menu items, which always occupy one line, form fields may occupy one or more lines (rows). Fields that occupy one line may be justified left, right, center, or not at all.

SYNTAX

```
int set_field_just (field, justification)  
FIELD * field;  
int justification;  
  
int field_just (field)  
FIELD * field;
```

Fields that occupy more than one line are not justified because the data entered typically extends into subsequent lines.

Manipulating Field Attributes

Setting the number of field columns (**cols**) and the minimum width or precision does not always determine where the data fits in the field—there may be excess character space before or after the data. Function **set_field_just()** lets you justify data in one of the following ways:

<code>NO_JUSTIFICATION</code>	- no justification processing (default)
<code>JUSTIFY_LEFT</code>	- left justify value in field
<code>JUSTIFY_RIGHT</code>	- right justify value in field
<code>JUSTIFY_CENTER</code>	- center value in the field

No matter what the justification, fields are automatically left justified as your end-user enters data and edits the field. Once field validation occurs upon the user's request to leave the field, ETI justifies the field as specified.

Note

By default, fields are not justified.

For instance, to left justify a name field and right justify an amount field, you can write:

```
FIELD * name, * amount;
set_field_just (name, JUSTIFY_LEFT); /* left justify a field */
set_field_just (amount, JUSTIFY_RIGHT); /* right justify a field */
```

If successful, **set_field_just()** returns `E_OK`. If not, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- bad justification

As with most other ETI functions, if one of these functions is passed a `NULL` field pointer, it assigns or fetches the system default. For instance, to change the system default from no justification to centering the value in its field, you write

```
set_field_just ( (FIELD *) 0, JUSTIFY_CENTER); /* set new default */
```

Setting the Field Foreground, Background, and Pad Character

The following functions enable you to set and read the pad character and the low-level ETI (**curses**) attributes associated with your field's foreground and background. The foreground attribute applies only to those field characters that represent data proper, while the background attribute applies to the entire field.

SYNTAX

```
int set_field_fore (field, attr)
FIELD * field;
chtype attr;
```

```
chtype field_fore (field)
FIELD * field;
```

```
int set_field_back (field, attr)
FIELD * field;
chtype attr;
```

```
chtype field_back (field)
FIELD * field;
```

```
int set_field_pad (field, pad)
FIELD * field;
int pad;
```

```
int field_pad (field)
FIELD * field;
```

The initial default for both the foreground and background are `A_NORMAL`. (See the section on attribute descriptions earlier in this guide or `curses(S)` in the *Programmer's Reference* for more on screen attributes.) The pad character is the character displayed wherever a blank occurs in the field value stored in field buffer 0.

As an example, to change the background of a field **total** to `A_UNDERLINE` and `A_STANDOUT`, you write

```
FIELD * total;

set_field_back (total, A_UNDERLINE | A_STANDOUT);
```

Setting the Field Foreground, Background, and Pad Character

If function `set_field_fore()` or `set_field_back()` encounter an error, they return one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- bad curses attribute

The function `set_field_pad()` sets the field's pad character. The default pad character is a blank. During form processing, pad characters in the field are translated to blanks in the field's value.

Note

Because ETI does not distinguish between system-generated pad characters and those entered as data, be sure to choose your pad character so as not to conflict with valid data.

To set the pad character for field **total** to an asterisk (*), you write

```
FIELD * total;  
  
set_field_pad (total, '*');
```

If successful, function `set_field_pad()` returns `E_OK`. If not, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- nonprintable pad character

As usual, you can change or access the ETI defaults. To change the default background to `A_UNDERLINE`, you write

```
set_field_back ((FIELD *) 0, A_UNDERLINE);
```


Some Helpful Features of Fields

ETI provides special features that promote development of a wide range of form applications. These include field buffers, field status flags, and field user pointers.

Setting and Reading Field Buffers

Recall that you set the number of additional buffers associated with a field upon its creation with `new_field()`. Buffer 0 holds the value of the field. The following functions let you store values in the buffers and later read them.

SYNTAX

```
int set_field_buffer (field, buffer, value)
FIELD * field;
int buffer;
char * value;

char * field_buffer (field, buffer)
FIELD * field;
int buffer
```

The parameter **buffer** should range from 0 through **nbuf**, where **nbuf** is the number of additional buffers in the `new_field()` call. All buffers besides 0 may be used to suit your application.

As an example, suppose your application kept a field's default value in field buffer 1. It could use the following code to reset the current field to its default value.

```
#define VAL_BUF      0
#define DFL_BUF      1

void reset_current (form)
FORM * form;
{
    /* set f to current field, described in
       section below, "Manipulating the Current Field" */
    FIELD * f = current_field (form);
    /* set field f to default value */
    set_field_buffer (f, VAL_BUF, field_buffer (f, DFL_BUF));
}
```

Some Helpful Features of Fields

If successful, `set_field_buffer()` returns `E_OK`. If not, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field pointer, null value, or buffer out of range

Function `field_buffer()`, however, returns `NULL` if its `field` pointer is `NULL` or `buffer` is out of range.

The function `field_buffer()` always returns the correct value if the field is not current. However, if the field is current, the function is sometimes inaccurate because data is not moved to field buffer 0 immediately upon entry. You may rest assured that `field_buffer()` is accurate on the current field if

- it is called from the field check validation routine, if any
- it is called from the form or field initialization or termination routines, if any
- it is called just after a `REQ_VALIDATION` request to the form driver

See the sections below, “Creating a Field Type with Validation Functions,” “Establishing Field and Form Initialization and Termination Routines,” and “Field Validation Requests,” for details on these routines.

Setting and Reading the Field Status

Every field has an associated status flag that is set whenever the field’s value (field buffer 0) changes. The following functions enable you to set and access this flag.

SYNTAX

```
int set_field_status (field, status)
FIELD * field;
int status;

int field_status (field)
FIELD * field;
```

The field status is `TRUE` if set or `FALSE` if cleared. By default, the field status is `FALSE` when the field is created.

These routines promote increased efficiency where processing need occur only if a field has been changed since some previous state. Two examples are undo operations and database updates. Function **update()** in Figure 10-33, for instance, loops through your field pointer array to save the data in each field if it has been changed (if its **field_status()** is TRUE).

Figure 10-33 Using the Field Status to Update a Database

```
void update (form)
FORM * form;
void save_field_data (f)
FIELD * f;
{
    char * data = field_buffer (f, 0); /* fetch data in field */

    /* save data */

}

{
    FIELD ** f = form_fields (form); /* fetch pointer to field
                                      pointer array */

    while (*f)
    {
        if (field_status (*f)) /* field data changed ? */
        {
            save_field_data (*f); /* yes, save new data */
            set_field_status (*f, FALSE); /* set field status back */
        }
        ++f;
    }
}
```

If successful, **set_field_status()** returns **E_OK**. If not, it returns the following:

E_SYSTEM_ERROR - system error

The initial ETI default field status is clear. As always, you can change the default by passing **set_field_status()** a NULL field pointer.

Some Helpful Features of Fields

Like the function `field_buffer()`, function `field_status()` always returns the correct value if the field is not current. However, if the field is current, the function is sometimes inaccurate because the status flag is not set immediately. You may rest assured that `field_status()` is accurate on the current field if

- it is called from the field check validation routine, if any
- it is called from the form or field initialization or termination routines, if any
- it is called just after a `REQ_VALIDATION` request to the form driver

See the sections below, “Creating a Field Type with Validation Functions,” “Establishing Field and Form Initialization and Termination Routines,” and “Field Validation Requests,” for details on these routines.

Setting and Fetching the Field User Pointer

As it does with panels and menus, ETI provides functions to manipulate an arbitrary pointer convenient for field data such as title strings, help messages, and the like.

SYNTAX

```
int set_field_userptr (field, userptr)
FIELD * field;
char * userptr;

char * field_userptr (field)
FIELD * field;
```

You can connect an application-defined structure to the field using this pointer. By default, the field user pointer is `NULL`.

Figure 10-34, for example, shows three routines that use these field functions:

- | | |
|------------------------|--|
| set_field_id() | allocates space for a struct ID to be associated with a field and calls set_field_userptr() to establish the field's pointer to it |
| free_field_id() | frees the space for the associated ID |
| find_match() | searches the names associated with all fields on the form to determine whether any of them match an arbitrary name passed to it |

Figure 10-34 Using the Field User Pointer to Match Items

```

#define match(a,b) (strcmp (a, b) == 0)

typedef struct
{
    int      type;
    char *   name;
}
    ID; /* to be hooked onto field userptr */

void set_field_id (f, type, name) /* associate type and name
                                with field f */
FIELD * f;
int type;
char * name;
{
    ID * id = (ID *) malloc (sizeof (ID)); /* allocate space,
                                           see malloc(S) */

    if (id) /* if space allocated */
    {
        id -> type = type; /* assign type and name */
        id -> name = name;
    }
    set_field_userptr (f, (char *) id); /* point to id */
}

void free_field_id (f) /* free id connected to field */
FIELD * f;
{
    x = (ID *) field_userptr (*f); /* fetch field user pointer */
    if (x)
        free (x);
}

FIELD * find_field (f, name) /* find field on form with name */
FORM * form;
char * name;
{
    FIELD ** f = form_fields (form); /* fetch pointer to form's
                                     field array */
    ID * x;

    while (*f) /* for each field in the form */
    {
        x = (ID *) field_userptr (*f);
        /* fetch ID associated with field */

        if (x && x -> name && match (name, x -> name))
            /* does its name match ? */
            break;
        ++f;
    }
    return *f; /* return field pointer of match or NULL */
}

```

Note that if a match is not found, `find_field()` returns a NULL field pointer. See the previous sections on panel and menu user pointers for more examples.

Some Helpful Features of Fields

If successful, **set_field_userptr()** returns **E_OK**. If not, it returns the following:

E_SYSTEM_ERROR - system error

To change the system default user pointer from **NULL** to one of your choice, you need only pass **set_field_userptr()** a **NULL** field pointer. Passing a **NULL** field pointer to **field_userptr()** returns the current default user pointer.

Manipulating Field Options

ETI provides several field options for controlling how data is entered and displayed in a field. The following functions let you set or clear these options or read their settings.

SYNTAX

```
int set_field_opts (field, opts)
FIELD * field;
OPTIONS opts;
```

```
OPTIONS field_opts (field)
FIELD * field;
```

options:

```
O_VISIBLE
O_ACTIVE
O_PUBLIC
O_EDIT
O_WRAP
O_BLANK
O_AUTOSKIP
O_NULLOK
O_PASSOK
```

Function **set_field_opts()** turns off all options that do not appear in its second argument. By default, all options are on.

The field options and their effects are as follows:

O_VISIBLE

determines field visibility. If this option is on, the field is displayed. If this option is off, it is erased. This option is useful for supporting pop-up fields, fields visible or not depending on another field's value.

O_ACTIVE

determines if a field is visited during form processing. If inactivated, the field is ignored during form processing. Inactive fields enable you to create field labels and other static form symbols or changeable symbols that are not affected during form processing. Examples of fields that change value but are not affected during form processing are row and column totals, as in a spreadsheet program. You can change field values using calls to **set_field_buffer()**.

Manipulating Field Options

- O_PUBLIC** determines how feedback is presented to the user as data is entered. The data in public fields is displayed as entered, while the data in non-public fields is not displayed at all. Further, in non-public fields, the cursor does not actually move across the field, but the forms subsystem internally maintains the cursor position relative to the field data. You can use non-public fields to implement password fields.
- O_EDIT** determines if field editing is permitted. By default, this option is on and a field may be edited. If the O_EDIT option is off, the field may be visited but not changed. Editing requests or attempts to enter data will fail. (REQ_PREV_CHOICE and REQ_NEXT_CHOICE requests, however, are honored, if they are defined for the field's type.) This is useful for creating fields for browsing such as scrollable help messages.
- O_WRAP** determines if word wrapping occurs at the end of each line of the field. If any character of the word does not fit on the line as it is entered, the entire word is automatically moved to the beginning of the next line, if there is one. If the O_WRAP option is off, the word is split between the two lines.
- O_BLANK** determines if the whole field is automatically erased when the end-user types a character in the first character position of the field before any character position has been changed. If the O_BLANK option is off, this does not occur.
- O_AUTOSKIP** determines how the field responds when it becomes full. Ordinarily, when a field is full, an automatic request to move to the next field on the form is generated. If, however, the O_AUTOSKIP option is off, your end-user remains at the end of the field.

O_NULLOK

determines how the field responds when your end-user tries to leave a blank field. By default, this option is on—when a field is blank, a request to leave the field is honored without validating the field. If, on the other hand, the O_NULLOK option is off, the validation procedure is applied to the blank field.

O_PASSOK

When this option is on, the field is checked for validity only if your end-user entered data into the field or edited it. If it is off, the validity check occurs whenever your user leaves the field, whether or not the field was changed. This is useful for fields whose validation function may change dynamically.

Remember that options are Boolean values. So to turn off option O_ACTIVE for field **f0** and to turn it on for field **f1**, you use the Boolean operators and write:

```
FIELD * f0, * f1;
```

```
set_field_opts (f0, field_opts (f0) & ~O_ACTIVE); /* turn option off */  
set_field_opts (f1, field_opts (f1) | O_ACTIVE); /* turn option on */
```

Note

Although you can change field option settings on posted forms, you cannot change option settings for the current field.

ETI also provides the following two functions which let you turn a field option on or off without using function **field_opts()**.

SYNTAX

```
int field_opts_on (field, opts)  
FIELD * field;  
OPTIONS opts;  
  
int field_opts_off (field, opts)  
FIELD * field;  
OPTIONS opts;
```


Manipulating Field Options

Unlike function `set_field_opts()`, these functions leave unnamed option settings intact.

As an example, the following code turns options `O_BLANK` and `O_AUTOSKIP` off for field `f0` and on for field `f1`:

```
FIELD * f0, * f1;

field_opts_off (f0, O_BLANK | O_AUTOSKIP); /* turn options off */

field_opts_on  (f1, O_BLANK | O_AUTOSKIP); /* turn options on  */
```

If successful, functions `set_field_opts()`, `field_opts_on()`, and `field_opts_off()` return `E_OK`. If not, they return the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_CURRENT</code>	- cannot change current field options

As usual, you can change the ETI default option settings by passing function `set_field_options()`, `field_opts_on()`, or `field_opts_off()` a `NULL` field pointer. Calling `field_opts()` with a `NULL` field pointer returns the system default.

Creating and Freeing Forms

Once you have established a set of fields and their attributes, you are ready to create a form to contain them.

SYNTAX

```
FORM * new_form (fields)
FIELD ** fields;

int free_form (form)
FORM * form;
```

The function **new_form()** takes as an argument a NULL-terminated, ordered array of **FIELD** pointers that define the fields on the form. The order of the field pointers determines the order in which the fields are visited during form driver processing discussed below.

As with the comparable ETI menu function **new_menu()**, function **new_form()** does not copy the array of field pointers. Instead, it saves the pointer to the array. Be sure not to change the array of field pointers once it has been passed to **new_form()**, until the form is freed by **free_form()** or the field array replaced by **set_form_fields()** described in the next section.

Fields passed to **new_form()** are connected to the resulting form.

Note

Fields may be connected to only one form at a time.

To connect fields to another form, you must first disconnect them using **free_form()** or **set_form_fields()**. If **fields** is NULL, the form is created but no fields are connected to it.

Creating and Freeing Forms

Unlike menus, ETI forms are logically divided into pages. Two functions enable you to mark a field that is to start a new page and to return a Boolean value indicating whether a given field does so.

SYNTAX

```
int set_new_page(field, bool)
FIELD * field;
int Bool;                /* TRUE or FALSE */

int new_page(field)
FIELD * field;
```

The initial system default value of `new_page()` is `FALSE`. This means that, unless specified with `set_new_page()`, each field is assumed to continue the current page.

Note

In general, you should make the size of each form page smaller than the form's window size.

If function `set_new_page()` executes successfully, it returns `E_OK`. If not, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	-system error
<code>E_CONNECTED</code>	-field connected to form

Figure 10-35 shows how to create a simple two-page form.

Figure 10-35 Creating a Form

```
FIELD * f[7];
FORM * form;

/* create fields as described in section above,
   "Creating and Freeing Fields" */

f[0] = new_field (...); /* 1st field on page 1 */
f[1] = new_field (...); /* 2nd field on page 1 */
f[2] = new_field (...); /* 3rd field on page 1 */
f[3] = new_field (...); /* 4th field on page 1 */

f[4] = new_field (...); /* 1st field on page 2 */
f[5] = new_field (...); /* 2nd field on page 2 */

f[6] = (FIELD *) 0; /* signal end of form */

set_new_page (f[4], TRUE); /* start new page with
                             fifth field f[4] */

form = new_form (f); /* create the form */
```

If successful, **new_form()** returns a pointer to the new form. If there is no memory available for the form or one of the given fields is connected to another form, **new_form()** returns NULL. Undefined results occur if the array of field pointers is not NULL-terminated.

The function **free_form()** disconnects all fields and frees any space allocated for the form. Its argument is a form pointer previously obtained from **new_form()**. The fields themselves are not automatically freed.

Note

You should free the fields comprising a form using **free_field()** only after you free their form using **free_form()**.

If successful, **free_form()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_POSTED	- form is posted

Posting forms is described below.

As with panel, item, menu, and field pointers, form pointers should not be used once they are freed. If they are, undefined results occur.

Manipulating Form Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A form attribute is any form feature whose value can be set or read by an appropriate ETI function. The set of fields connected to a form and the number of fields connected to it are examples of form attributes.

Changing and Fetching the Fields on an Existing Form

Once you create a form with one set of fields using `new_form()`, you can change the fields connected to it.

SYNTAX

```
int set_form_fields (form, fields)
FORM * form;
FIELD ** fields;

FIELD ** form_fields (form)
FORM * form;
```

Like `new_form()`, function `set_form_fields()` takes as an argument a NULL-terminated, ordered array of `FIELD` pointers that define the fields on the form and determine the order in which the fields are visited during form driver processing.

When `set_form_fields()` is called, the fields previously connected to the form are disconnected from it (but not freed) before the new fields are connected.

Like any set of fields connected to a form, the new fields cannot be passed to other forms while they are connected to the given form. You must first disconnect them by calling `free_form()` or again calling `set_form_fields()`.

Manipulating Form Attributes

There are two ways to disconnect the fields associated with a form without connecting another set of fields to the form:

- you can call **free_form()**
- you can call **set_form_fields()** with **fields** set to **NULL**

The first method frees the space allocated for the form, whereas the second does not.

To change the fields associated with **form** to those referenced in array pointer **newfields**, you can write

```
FORM * form;
FIELD ** newfields;

set_form_fields (form, newfields); /* associate new set of
                                   fields with form */
```

If function **set_form_fields()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_POSTED	- form is posted
E_CONNECTED	- connected field

Posting forms is discussed in the section below, "Posting and Unposting Forms."

The function **form_fields()** returns the array of field pointers defining the form's fields. The function returns **NULL** if no fields are connected to the form or the form pointer is **NULL**.

Manipulating Form Attributes

Counting the Number of Fields

The following function returns the number of fields connected to the given form.

```
SYNTAX  
  
int field_count (form)  
FORM * form;
```

If **form** is NULL, **field_count()** returns -1.

As an example, consider the following routine, which determines whether your user is on the last field of the form as numbered in the field pointer array:

```
int on_last_field (form)  
FORM * form;  
{  
    /* fetch number of last field */  
  
    int lastindex = field_count (form) - 1;  
  
    /* determine whether number of current field is the same */  
  
    return field_index (current_field (form)) == lastindex;  
}
```

Note the use of functions **field_index()** and **current_field()**, described below in the section “Manipulating the Current Field.”

Changing ETI Form Default Attributes

During form initialization using **new_form()**, all form attributes are assigned default values. As you can with menu attributes, you can change these default attribute values by calling the appropriate function with a NULL form pointer as its first argument. All subsequent forms created using **new_form()** will then have the new default attribute value. However, forms created before the change to the current default value will retain the initial values of their attributes. Several examples of changing default values occur throughout the rest of this chapter.

Displaying Forms

In general, to display a form, you determine the form dimensions, optionally associate a window and subwindow with the form, post the form, and refresh the screen.

Determining the Dimensions of Forms

Every form is associated with a window and subwindow.

Note

By default, (C) the form window is NULL, which by convention means that ETI uses **stdscr** as the form window; and (S) the form subwindow is NULL, which means that ETI uses the form window as the form subwindow.

Windows are used to create borders, titles, and the like. Before ETI posts a form, it must determine the sizes of its window and subwindow.

To determine the minimum window or subwindow size for a form, ETI considers the following:

- the number of rows and columns for each field
- the starting position (upper left corner) of each field within the form subwindow

By automatically fetching this information previously established by calls to **new_field()**, function **scale_form()** saves you the effort of calculating the size of your form subwindow.

Displaying Forms

Scaling the Form

Considering the above information, this function returns the minimum window size necessary for containing the form.

SYNTAX

```
int scale_form (form, rows, cols)
FORM * form;
int * rows;
int * cols;
```

Because function **scale_form()** must return more than one value (namely, the minimum number of rows and columns for the form) and C passes parameters “by value” only, the arguments of **scale_form()** are pointers. The pointer arguments **rows** and **cols** point to locations used to return the minimum number of rows and columns for the form.

Note

You should call **scale_menu()** only after the form’s fields have been connected to the form using **new_form()** or **set_form_fields()**.

As an example, to return the minimum (sub)window size for form **f** in variables **rows** and **cols**, you can write the following:

```
FORM * form;
int rows, cols;

/* create fields
create form */

/* determine minimum row and column size */
scale_form (form, &rows, &cols);

/* create form subwindow,
as described in next section */
```

If function **scale_form()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_NOT_CONNECTED	- no fields connected to the form

Associating Windows and Subwindows with a Form

Remember that two windows are associated with every form—the form window and the form subwindow. The following functions assign windows and subwindows to forms and fetch those previously assigned to them.

SYNTAX

```
int set_form_win (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_win (form)
FORM * form;

int set_form_sub (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_sub (form)
FORM * form;
```

These functions enable you to place stylistic borders, titles, and other decoration around a form.

Note

Remember that if the form window is NULL (the default), ETI uses **stdscr**. If the form subwindow is NULL (the default), ETI uses the form window so you need not use functions **set_form_win()** or **set_form_sub()** at all.

If you do not want to use **stdscr**, you should create a window and a subwindow for every form. ETI automatically writes all low-level ETI (**curses**) output of the form proper on the form subwindow. If you want further output (such as borders, titles, or static messages), you should write it on the form window. However, you need not write any further output at all.

Displaying Forms

Note

Be sure to apply all low-level ETI (**curses(S)**) command output and refresh operations to your form's window, not its subwindow.

Figure 10-36 diagrams the relationship between ETI Form functions, your application program, and its form window and subwindow.

Figure 10-36 Form Functions Write to Subwindow, Application to Window

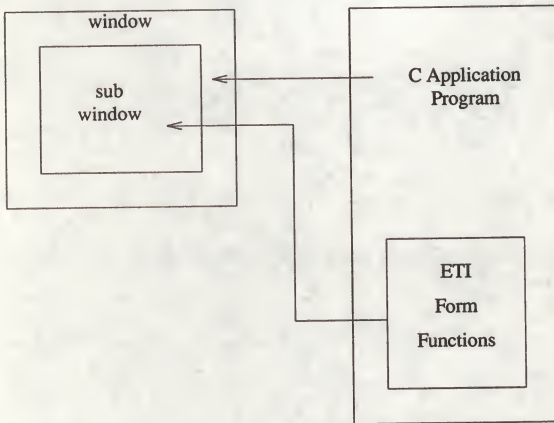


Figure 10-37 shows how to create a form with a border of the terminal's default vertical and horizontal characters.

Figure 10-37 Creating a Border Around a Form

```

/* create window 4 characters larger than form dimensions
with top left corner at (0, 0). subwindow is positioned
at (2, 2) relative to the form window origin with dimensions
equal to the form dimensions. */

FORM * f;
WINDOW * w;
int rows, cols;

scale_form (f, &rows, &cols); /* get dimensions of form */

if (w = newwin (rows+4, cols+4, 0, 0))
{
    set_form_win (f, w); /* associate window and subwindow
                           with form */

    set_form_sub (f, derwin (w, rows, cols, 2, 2));

    box (w, 0, 0); /* create border */
}

```

Function **scale_form()** sets the values of the variables **rows** and **cols**, which provide the form dimensions without the border. Adding four to the dimensions of the form window clearly sets off the form border from the fields of the form (the form proper).

If functions **set_form_win()** or **set_form_sub()** encounter an error, they return one of the following:

E_SYSTEM_ERROR	- system error
E_POSTED	- form is posted

As usual, you can change the default form window or subwindow. For instance, you can change the default form window from **stdscr** to a window **w** by passing a NULL form pointer, as follows:

```

int rows, cols, firstrow, firstcol;

/* create form window */

WINDOW * w = newwin (rows, cols, firstrow, firstcol);

set_form_win((FORM *)0, w); /* change default form window to w */

```

Note that if you later change a posted form by writing directly to its window, before continuing you must reposition the form window cursor using **pos_form_cursor()**. See the section below, “Positioning the Form Cursor.”

Displaying Forms

Posting and Unposting Forms

When you have created a form and its window and subwindow, you are ready to post it. To post a form is to display it on the form's subwindow; to unpost a form is to erase it from the form's subwindow.

SYNTAX

```
int post_form (form)
FORM * form;

int unpost_form (form)
FORM * form;
```

Unposting a form does not remove its data structure from memory.

Note

To post a form, be sure that you have connected fields to it first.

Figure 10-38 uses two application routines, **display_form()** and **erase_form()**, to show how you might post and later unpost a form. The code builds on that used previously in Figure 10-37 to create the form's window and subwindow.

Figure 10-38 Posting and Unposting a Form

```
static void display_form (f)      /* create form windows and post */
FORM * f;
{
    WINDOW * w;
    int rows;
    int cols;

    scale_form (f, &rows, &cols); /* get dimensions of form */

    /* create form window as in Figure 10-37 */

    if (w = newwin (rows+4, cols+4, 0, 0))
    {
        set_form_win (f, w);
        set_form_sub (f, derwin (w, rows, cols, 2, 2));
        box (w, 0, 0);
        keypad (w, 1);
    }
    else
        /* error routine in previous section
        * "ETI Low-Level Interface (curses)
        * to High-Level Functions" */
        error ("error return from newwin", NULL);

    if (post_form (f) != E_OK)      /* post form */

        error ("error return from post_form", NULL);
    else
        wrefresh (w);
}

static void erase_form (f)      /* unpost and delete form windows */
FORM * f;
{
    WINDOW * w = form_win (f);
    WINDOW * s = form_sub (f);

    unpost_form (f);      /* unpost form */
    werase (w);           /* erase form window */
    wrefresh (w);         /* refresh screen */
    delwin (s);           /* delete form windows */
    delwin (w);
}
```

Displaying Forms

If successful, function **post_form()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_POSTED	- form is already posted
E_NOT_CONNECTED	- no connected fields
E_NO_ROOM	- form does not fit in subwindow

If successful, the function **unpost_form()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_NOT_POSTED	- form is not posted
E_BAD_STATE	- called from init/term function

The initialization and termination routines are discussed in the next section.

Form Driver Processing

Like the function `menu_driver()` for the menu subsystem, function `form_driver()` is the workhorse of the form system. Once the form is posted, the form driver handles all interaction with your end-user. The form driver responds to

- field navigation requests
- page navigation requests
- field editing requests
- data entry
- field validation requests

Your application passes a character to the form driver for processing and evaluates the results.

SYNTAX

```
int form_driver (form, c)
FORM * form;
int c;
```

As with menu processing, to enable the form driver to process your end-users' requests, you must write an input key virtualization routine. This routine defines a correspondence between input keys, control characters, and escape sequences on the one hand and ETI form requests on the other. The routine returns a specific form request or application command that the form driver can process. Upon return from the form driver, your application can check if the input was processed appropriately. If not, it can specify actions to be taken. These may include terminating interaction with the form, responding to help requests, generating an error message, and so on.

Defining the Virtual Key Mapping

For a sample virtual key mapping, consider Figure 10-39, which contains the application-defined function `get_request()`. Most of the values returned by `get_request()` are ETI form requests defined in header file `form.h` and described in the next section. The other values returned (in this example, only value `QUIT`) are defined by the application program treated in the later section "Calling the Form Driver."

Figure 10-39 A Sample Key Virtualization Routine

/* The following key mapping is defined by `get_request()`.
Note that `^X` represents the character control-X.

<code>^Q</code>	- end form processing
<code>^F</code>	- move to next page
<code>^B</code>	- move to previous page
<code>^N</code>	- move to next field
<code>^P</code>	- move to previous field
home key	- move to first field
home down	- move to last field
<code>^L</code>	- move left to field
<code>^R</code>	- move right to field
<code>^U</code>	- move up to field
<code>^D</code>	- move down to field
<code>^W</code>	- move to next word
<code>^T</code>	- move to previous word
<code>^S</code>	- move to beginning of field data
<code>^E</code>	- move to end of field data
left arrow	- move left in field
right arrow	- move right in field
down arrow	- move down in field
up arrow	- move up in field
<code>^M</code> <CR>	- enter new line
<code>^I</code>	- insert blank character
<code>^O</code>	- insert blank line
<code>^V</code>	- delete character
<code>^H</code> <BS>	- delete previous character
<code>^Y</code>	- delete line
<code>^G</code>	- delete word
<code>^C</code>	- clear to end of line
<code>^K</code>	- clear to end of field
<code>^X</code>	- clear entire field
<code>^A</code>	- request next field choice
<code>^Z</code>	- request previous field choice
<code>ESC</code>	- toggle between insert and overlay mode

define application commands */

(Continued on next page.)

Figure 10-39 A Sample Key Virtualization Routine

(Continued)

```

#define QUIT      (MAX_COMMAND + 1)

static int get_request (w)      /* virtual key mapping */
WINDOW * w;
{
    static int    mode    = REQ_INS_MODE;
    int          c        = wgetch (w); /* read a character */

    switch (c)
    {
        case 0x11: /* ^Q */      return    QUIT;

        case 0x06: /* ^F */      return    REQ_NEXT_PAGE;
        case 0x02: /* ^B */      return    REQ_PREV_PAGE;
        case 0x0e: /* ^N */      return    REQ_NEXT_FIELD;
        case 0x10: /* ^P */      return    REQ_PREV_FIELD;
        case KEY_HOME:          return    REQ_FIRST_FIELD;
        case KEY_LL:           return    REQ_LAST_FIELD;
        case 0x0c: /* ^L */      return    REQ_LEFT_FIELD;
        case 0x12: /* ^R */      return    REQ_RIGHT_FIELD;
        case 0x15: /* ^U */      return    REQ_UP_FIELD;
        case 0x04: /* ^D */      return    REQ_DOWN_FIELD;
        case 0x17: /* ^W */      return    REQ_NEXT_WORD;
        case 0x14: /* ^T */      return    REQ_PREV_WORD;
        case 0x13: /* ^S */      return    REQ_BEG_FIELD;
        case 0x05: /* ^E */      return    REQ_END_FIELD;
        case KEY_LEFT:          return    REQ_LEFT_CHAR;
        case KEY_RIGHT:         return    REQ_RIGHT_CHAR;
        case KEY_DOWN:          return    REQ_DOWN_CHAR;
        case KEY_UP:            return    REQ_UP_CHAR;
        case 0x0d: /* ^M */      return    REQ_NEW_LINE;
        case 0x09: /* ^I */      return    REQ_INS_CHAR;
        case 0x0f: /* ^O */      return    REQ_INS_LINE;
        case 0x16: /* ^V */      return    REQ_DEL_CHAR;
        case 0x08: /* ^H */      return    REQ_DEL_PREV;
        case 0x19: /* ^Y */      return    REQ_DEL_LINE;
        case 0x07: /* ^G */      return    REQ_DEL_WORD;
        case 0x03: /* ^C */      return    REQ_CLR_EOL;
        case 0x0b: /* ^K */      return    REQ_CLR_EOF;
        case 0x18: /* ^X */      return    REQ_CLR_FIELD;
        case 0x01: /* ^A */      return    REQ_NEXT_CHOICE;
        case 0x1a: /* ^Z */      return    REQ_PREV_CHOICE;
        case 0x1b: /* ^ESC */
            if (mode == REQ_INS_MODE)
                return mode = REQ_OVL_MODE;
            else
                return mode = REQ_INS_MODE;
    }
    return c;
}

```


Form Driver Processing

In `get_request()`, only a subset of the requests are defined so that the requests your end-user can make are limited. If you like, you can also map two or more keys onto one request. This is helpful where some terminals lack one of the keys in question. In that case, the user can press the other key to the same effect.

Function `get_request()` first sets the data entry mode for the end-user. Here it is set initially to insert mode. The last case statement in the routine enables your end-user to press the escape key ESC to switch to overlay mode. Both modes are discussed in the “Field Editing Requests” section below.

Next, `get_request()` calls `wgetch()` to read a character entered by the user. The `switch()` statement maps the character read onto a specific application command or form request. The application command QUIT appears here as the first case; the other cases map characters onto form requests. Any character that is not an application command or form request is simply returned unchanged—it is treated as data being entered into the current field.

Note that this key mapping assumes your end-user will be using a terminal with arrow keys (`KEY_LEFT`, `KEY_RIGHT`, `KEY_UP`, `KEY_DOWN`), a home key (`KEY_HOME`), and a home down key (`KEY_LL`).

ETI Form Requests

The ETI form subsystem places the following requests at your application program’s disposal.

Page Navigation Requests

These requests enable your end-user to navigate or move from page to page on a multi-page form.

<code>REQ_NEXT_PAGE</code>	- move to next page
<code>REQ_PREV_PAGE</code>	- move to previous page
<code>REQ_FIRST_PAGE</code>	- move to first page
<code>REQ_LAST_PAGE</code>	- move to last page

Page navigation requests are cyclic so that

- the REQ_NEXT_PAGE request from the last page moves to the first page
- the REQ_PREV_PAGE from the first page moves to the last page

Inter-Field Navigation Requests on the Current Page

These requests enable your end-user to move from field to field on the current page of a single form.

REQ_NEXT_FIELD	- move to next field
REQ_PREV_FIELD	- move to previous field
REQ_FIRST_FIELD	- move to first field
REQ_LAST_FIELD	- move to last field
REQ_SNEXT_FIELD	- move to sorted next field
REQ_SPREV_FIELD	- move to sorted previous field
REQ_SFIRST_FIELD	- move to sorted first field
REQ_SLAST_FIELD	- move to sorted last field
REQ_LEFT_FIELD	- move left to field
REQ_RIGHT_FIELD	- move right to field
REQ_UP_FIELD	- move up to field
REQ_DOWN_FIELD	- move down to field

All field navigation requests are cyclic on the current page so that

- the REQ_NEXT_FIELD request from the last field on a page moves to the first field on that page
- the REQ_PREV_FIELD request from the first field on a page moves to the last field on that page

and so forth. The order of the fields in the field array passed to **new form()** determines the order in which the fields are visited using the REQ_NEXT_FIELD, REQ_PREV_FIELD, REQ_FIRST_FIELD, and REQ_LAST_FIELD requests.

Note

Remember that the order of fields in the form array is simply the order in which fields are processed during form processing. This order bears no necessary relation to the order of the fields as they are displayed on the form page.

Form Driver Processing

Your end-user may also move from field to field on the form page in row-major order—left to right, top to bottom. To do so, you use the `REQ_SNEXT_FIELD`, `REQ_SPREV_FIELD`, `REQ_SFIRST_FIELD`, and `REQ_SLAST_FIELD` requests.

Finally, your end-user can move about in different directions using the `REQ_LEFT_FIELD`, `REQ_RIGHT_FIELD`, `REQ_UP_FIELD`, and `REQ_DOWN_FIELD` requests. Note that the first character (top left corner) of the field is used to determine where the field is located relative to other fields. This means, for example, that a multi-line field whose first character is on the second row of a form is not on the same row as a field whose first character is on the third row of a form even though the multi-line field may extend below the third row.

Intra-Field Navigation Requests

These requests let your end-user move about inside a field. They may generate implicit scrolling operations on scrollable fields.

<code>REQ_NEXT_CHAR</code>	- move to next character in field
<code>REQ_PREV_CHAR</code>	- move to previous character in field
<code>REQ_NEXT_LINE</code>	- move to next line in field
<code>REQ_PREV_LINE</code>	- move to previous line in field
<code>REQ_NEXT_WORD</code>	- move to next word in field
<code>REQ_PREV_WORD</code>	- move to previous word in field
<code>REQ_BEG_FIELD</code>	- move to beginning of field
<code>REQ_END_FIELD</code>	- move after last character in field
<code>REQ_BEG_LINE</code>	- move to beginning of line
<code>REQ_END_LINE</code>	- move after last character in line
<code>REQ_LEFT_CHAR</code>	- move left in field
<code>REQ_RIGHT_CHAR</code>	- move right in field
<code>REQ_UP_CHAR</code>	- move up in field
<code>REQ_DOWN_CHAR</code>	- move down in field

The effect of these requests is as follows:

- The `REQ_NEXT_CHAR` and `REQ_PREV_CHAR` requests step forward and backward through the field.
- The `REQ_NEXT_LINE` and `REQ_PREV_LINE` requests move the cursor to the beginning of the next and previous line.
- The `REQ_NEXT_WORD` and `REQ_PREV_WORD` requests move the cursor to the beginning of the next or previous word.

- The `REQ_BEG_FIELD` places the cursor at the first non-pad character in the field. The `REQ_END_FIELD` request places the cursor after the last non-pad character in the field. This lets the user easily add characters to the field. If there is no room, it returns the cursor to the start of the field.
- The `REQ_BEG_LINE` request places the cursor at the first non-pad character in the current line of the field. The `REQ_END_LINE` request places the cursor after the last non-pad character in the current line. If there is no room, it returns the cursor to the start of the line.
- The `REQ_LEFT_CHAR`, `REQ_RIGHT_CHAR`, `REQ_UP_CHAR`, and `REQ_DOWN_CHAR` requests move one character position in the stated direction.

Field Editing Requests

These requests set the editing mode—insert or overlay.

<code>REQ_INS_MODE</code>	- begin insert mode
<code>REQ_OVL_MODE</code>	- begin overlay mode

In insert mode (the default), all text is inserted at the current cursor position, while all existing text starting at the current cursor position is moved to the right. In overlay mode, text entered by your end-user overlays (replaces) existing text in the field. In both modes, the cursor is advanced one character position as each character is entered.

The following requests provide a complete set of field editing requests.

<code>REQ_NEW_LINE</code>	- new line request
<code>REQ_INS_CHAR</code>	- insert blank character at cursor
<code>REQ_INS_LINE</code>	- insert blank line at cursor
<code>REQ_DEL_CHAR</code>	- delete character at cursor
<code>REQ_DEL_PREV</code>	- delete character before cursor
<code>REQ_DEL_LINE</code>	- delete line at cursor
<code>REQ_DEL_WORD</code>	- delete word at cursor
<code>REQ_CLR_EOL</code>	- clear to end of line
<code>REQ_CLR_EOF</code>	- clear to end of field
<code>REQ_CLR_FIELD</code>	- clear entire field

The effects of `REQ_NEW_LINE` and `REQ_DEL_PREV` requests depend on several factors such as the current mode (insert or overlay) and the cursor position within the field.

Form Driver Processing

- The effects of REQ_NEW_LINE are as follows:
 - In insert mode—if the cursor is at the beginning of a field or on the last line of a field, the REQ_NEW_LINE request acts like a REQ_NEXT_FIELD request. Otherwise, the REQ_NEW_LINE request inserts a new line after the current line and moves the text on the current line starting at the cursor position to the beginning of the new line. The cursor is moved to the beginning of the new line.
 - In overlay mode—if the cursor is at the beginning of a field, the REQ_NEW_LINE request acts like a REQ_NEXT_FIELD request. If the cursor is on the last line of a field, the REQ_NEW_LINE request erases all data from the cursor position to the end of the line and satisfies a REQ_NEXT_FIELD request. Otherwise, the REQ_NEW_LINE request erases all data from the cursor position to the end of the line and moves the cursor to the beginning of the next line.
- The effects of the REQ_DEL_PREV request is as follows:
 - In insert mode—if the cursor is at the beginning of a field, the REQ_DEL_PREV request behaves like a REQ_PREV_FIELD request. If the cursor is at the beginning of a line other than the first and the text on that line will fit at the end of the preceding line, the text is moved and the current line is deleted. Otherwise, the REQ_DEL_PREV request simply deletes the previous character.
 - In overlay mode—if the cursor is positioned at the beginning of a field, the REQ_DEL_PREV request behaves like a REQ_PREV_FIELD request. Otherwise, the REQ_DEL_PREV request simply deletes the previous character.

Because the requests REQ_NEW_LINE and REQ_DEL_PREV automatically do a request REQ_NEXT_FIELD or REQ_PREV_FIELD as described, they are said to be overloaded field editing requests. See the remarks on options O_NL_OVERLOAD and O_BS_OVERLOAD in the section below, “Setting and Fetching Form Options.”

Scrolling Requests

Remember that you specified the number of offscreen rows of a field, if any, as an argument to **new_field()** when the field was created. The following requests enable your program to scroll through fields to display this offscreen data.

REQ_SCR_FLINE	- scroll field forward a line
REQ_SCR_BLINE	- scroll field backward a line
REQ_SCR_FPAGE	- scroll field forward a page
REQ_SCR_BPAGE	- scroll field backward a page

In addition, intra-field navigation requests may generate implicit scrolling on scrollable fields. See the section above, “Intra-Field Navigation Requests.”

Field Validation Requests

This request supports field validation for those field types that have it.

REQ_VALIDATION	- validate current field
----------------	--------------------------

Note

In general, the ETI form driver automatically performs validation on a field before the user leaves it. (If your user leaves a field, it is valid.) However, before your user terminates interaction with the form, you should make the REQ_VALIDATION request to validate the current field.

Recall that on current fields, the values returned by functions **field_buffer()** and **field_status()** are sometimes inaccurate. (See the sections above, “Setting and Reading the Field Buffers” and “Setting and Reading the Field Status.”) If, however, you make request REQ_VALIDATION immediately before calling these functions, you can be sure that the values they return are accurate—they agree with what your end-user has entered and appears on the screen.

Form Driver Processing

Choice Requests

The following requests enable your user to request the next or previous value of a field type.

REQ_NEXT_CHOICE	- display next field choice
REQ_PREV_CHOICE	- display previous field choice

TYPE_ENUM is the only generic field type that supports these choice requests. In addition, programmer-defined field types may support these requests. See the section above, "Setting Field Type to Ensure Validation," and the section below, "Creating and Manipulating Programmer-Defined Field Types," for information on these field types.

Application-Defined Commands

Form requests are implemented as integers above the low-level ETI (**urses**) maximum key value KEY_MAX. A symbolic constant MAX_COMMAND is provided so applications can implement their own commands without conflicting with the ETI form or menu subsystems. All ETI system form requests are greater than KEY_MAX and less than or equal to MAX_COMMAND. You should set your application-defined commands to an integer greater than MAX_COMMAND.

Calling the Form Driver

The ETI form driver works very much like the ETI menu driver. As soon as the form driver receives a request, it checks if it is an ETI form request. If so, it performs the request and reports the results. If the request is not an ETI form request, the form driver checks if the character is data, i.e., a printable ascii character. If it is, it enters the character at the current position in the current field. If the character is not recognized as a form request or data, the form driver assumes the character is an application-defined command and returns E_UNKNOWN_COMMAND.

To illustrate a sample design for calling the form driver, we will consider a program that permits interaction with a sweepstakes entry form reproduced in Figure 10-40.

Figure 10-40 Sweepstakes Form Output

Sweepstakes Entry Form		
Last Name	First	Middle
_____	_____	_____
Comments		

You have already seen much of the sweepstakes program in previous examples. Figure 10-41 shows its remaining routines.

Form Driver Processing

Figure 10-41 An Example of Form Driver Usage

```
/* This program displays a sweepstakes entry form. */

#include <string.h>
#include <form.h>

static void start_curses() /* see the section above,
    "ETI Low Level Interface (curses)
    to High-Level Functions" */

static void display_form (f) /* create form windows and post */
    /* see Figure 10-38 for details */

static void erase_form (f) /* unpost and delete form windows */
    /* see Figure 10-38 for details */

/* define application commands */

#define QUIT (MAX_COMMAND + 1)

static int get_request (w) /* virtual key mapping
    see Figure 10-39 */

static int my_driver (form, c) /* handle application commands */
FORM * form;
int c;
{
    switch (c)
    {
        case QUIT:

            /* validate current field */

            if (form_driver (form, REQ_VALIDATION) == E_OK)
                return TRUE;
            break;
    }
    beep (); /* signal error */
    return FALSE;
}
```

(Continued on next page.)

Figure 10-41 An Example of Form Driver Usage*(Continued)*

```

main (argc, argv)
int argc;
char * argv[];
{
    WINDOW *   w;
    FORM *     form;
    FIELD **   f;
    FIELD **   make_fields ();
    void       free_fields ();
    int        c, done = FALSE;

    PGM = argv[0];

    if (! (form = new_form (make_fields ())))
        error ("error return from new_form", NULL);

    start_curses ();
    display_form (form);

    /* interact with user */

    w = form_win (form);

    while (! done)
    {
        switch (form_driver (form, c = get_request (w)))
        {
            case E_OK:
                break;
            case E_UNKNOWN_COMMAND:
                done = my_driver (form, c);
                break;
            default:
                beep ();    /* signal error */
                break;
        }
    }

    /* terminate form processing */

    erase_form (form);
    end_curses ();
    f = form_fields (form);
    free_form (form);
    free_fields (f);
    exit (0);
}

```

(Continued on next page.)

Figure 10-41 An Example of Form Driver Usage

(Continued)

```
typedef FIELD *      (* PF_field ) ();

typedef struct      /* define struct for creation */
{
    PF_field  type; /* field constructor */
    int       rows; /* number of rows    */
    int       cols; /* number of columns */
    int       frow; /* first row      */
    int       fcol; /* first column   */
    char *    v;    /* field value    */
}

FIELD_RECORD;

static FIELD * LABEL (x)      /* create a LABEL field */
FIELD_RECORD * x;
{
    FIELD * f = new_field (1, strlen (x->v), x->frow, x->fcol, 0, 0);

    if (f)
    {
        set_field_buffer (f, 0, x->v);
        field_opts_off (f, O_ACTIVE);
    }
    return f;
}

static FIELD * STRING (x)     /* create a STRING field */
FIELD_RECORD * x;
{
    FIELD * f = new_field (x->rows, x->cols, x->frow, x->fcol, 0, 0);

    if (f)
        set_field_back (f, A_UNDERLINE);
    return f;
}

/* field definitions */

static FIELD_RECORD F [] =
{
    LABEL,      0, 0, 0, 11, "Sweepstakes Entry Form",
    LABEL,      0, 0, 2, 0, "Last Name",
    LABEL,      0, 0, 2, 20, "First",
    LABEL,      0, 0, 2, 34, "Middle",
    LABEL,      0, 0, 5, 0, "Comments",
    STRING,     1, 18, 3, 0, (char *) 0,
    STRING,     1, 12, 3, 20, (char *) 0,
    STRING,     1, 12, 3, 34, (char *) 0,
    STRING,     4, 46, 6, 0, (char *) 0,
    (PF_field) 0, 0, 0, 0, 0, (char *) 0,
};
```

(Continued on next page.)

Figure 10-41 An Example of Form Driver Usage

(Continued)

```

#define MAX_FIELD      512

static FIELD *        fields [MAX_FIELD + 1]; /* field buffer */

static FIELD ** make_fields ()    /* create the fields */
{
    FIELD ** f = fields;
    int i;

    for (i = 0; i < MAX_FIELD && F[i].type; ++i, ++f)
        *f = (* F[i].type) (& F[i]);

    *f = (FIELD *) 0;
    return fields;
}

static void free_fields (f)      /* free the fields */
FIELD ** f;
{
    while (*f)
        free_field (*f++);
}

```

Function **main()** first calls an application-defined routine **make_fields()** to create the fields and **new_form()** to create the form. Routine **make_fields()** offers a somewhat different way to create fields from that used in the example in Figure 10-29. (Array **F** holds the string labels and field sizes; it can be changed so that **make_fields()** can create any form.) Function **main()** then initializes **curses** using **start_curses()** and displays the form using **display_form()**.

In its **while** loop, **main()** repeatedly calls **form_driver()** with the character returned by **get_request()**. If the form driver does not recognize the character as a request or data, it returns **E_UNKNOWN_COMMAND**, whereupon the application-defined routine **my_driver()** is called with the same character. Routine **my_driver()** processes the application-defined commands. In this example, there is only one, **QUIT**. Note how this request automatically calls the form driver again, now with the **REQ_VALIDATION** request. Remember that this request is necessary to ensure that current field validation occurs before your end-user leaves the form. If validation is successful, **my_driver()** returns **TRUE**. In turn, this sets **done** to **TRUE**, and the **while** loop is exited.

Finally, **main()** erases the form, terminates low-level ETI (**curses**), frees the form and its fields, and exits the program.

Form Driver Processing

This example is typical, but it is only one of many ways you can structure an application. ETI's flexibility lets you use it over a wide range of applications.

Like other ETI routines that return an **int**, the form driver returns **E_OK** if it recognizes and processes the input character argument. If it encounters an error, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	-system error
<code>E_BAD_ARGUMENT</code>	-null form pointer
<code>E_BAD_STATE</code>	-called from init/term routines
<code>E_NOT_POSTED</code>	-form is not posted
<code>E_UNKNOWN_COMMAND</code>	-unknown command
<code>E_REQUEST_DENIED</code>	-recognized request failed
<code>E_INVALID_FIELD</code>	-failed field validation

Note

Like the menu driver, the form driver may not be called from any of the initialization or termination routines described next. Any attempt to do so returns **E_BAD_STATE**.

Establishing Field and Form Initialization and Termination Routines

As with the menu driver, you may sometimes want the form driver to execute a specific routine whenever the current field or form changes. The following routines let you do this.

SYNTAX

```
typedef void (* PTF_void ) ();

int set_form_init (form, func)
FORM * form;
PTF_void func

PTF_void form_init (form)
FORM * form;

int set_form_term (form, func)
FORM * form;
PTF_void func;

PTF_void form_term (form)
FORM * form;

int set_field_init (form, func)
FORM * form;
PTF_void func

PTF_void field_init (form)
FORM * form;

int set_field_term (form, func)
FORM * form;
PTF_void func;

PTF_void field_term (form)
FORM * form;
```

The argument **func** is a pointer to the specific function you want executed by the form driver. This application-defined function itself takes a form pointer as an argument.

As with menus, if you want your application to execute a routine at one of the initialization or termination points listed below, you should call the appropriate form initialization or termination routine at the start of your program. If you do not want a specific function called in these cases, you may refrain from calling these routines altogether.

Form Driver Processing

Function `set_form_init()`

The argument **func** to this function is automatically called by the form driver

- when the form is posted
- just after every form page operation, i.e., after the page changes on a posted form

Function `set_field_init()`

The argument **func** to this function is automatically called by the form driver

- when the form is posted
- just after a field change operation, i.e., every time the current field changes on a posted form

Function `set_field_term()`

The argument **func** to this function is automatically called by the form driver

- just after the field is validated, i.e., just before the current field changes on a posted form
- when the form is unposted

Function `set_form_term()`

The argument **func** to this function is automatically called by the form driver

- just before every form page operation, i.e., just before the page changes on a posted form
- when the form is unposted

To see more precisely when the initialization and termination routines may be executed, note that your form page and current field can be changed in the following circumstances:

- Both the form page and the current field may be changed automatically by the form driver in response to a user's request.
- The form page may be changed when the current field is changed using `set_current_field()`.
- The current field is changed when the page is changed using `set_form_page()`.

Note

All of these initialization and termination functions are NULL by default. This means that no function need be called.

These functions promote common operations, such as row or column total updates, display of previously invisible fields, activation of previously inactive fields, and more. As an example, Figure 10-42 shows a field termination routine `update_total()`, which dynamically adjusts a column total field whenever a row field value changes. Function `main()` calls `set_field_term()` to establish `update_total()` as the field termination routine.

Form Driver Processing

Figure 10-42 Sample Termination Routine that Updates a Column Total

```
#include <form.h>

void update_total (form)
FORM * form;
{
    FIELD ** f = form_fields (form);
    char buf[80];
    double total, atof(); /* atof() converts string to float */

    switch (field_index (current_field (form)))
    {
        case ROW_1:
        case ROW_2:
        case ROW_3:

            /* field_buffer() returns field's value as string,
               which atof() converts to float */

            total = atof (field_buffer (f[ROW_1], 0)) + /*calculate total*/
                    atof (field_buffer (f[ROW_2], 0)) +
                    atof (field_buffer (f[ROW_3], 0));

            sprintf (buf, "%.2f", total);
            set_field_buffer (f[TOTAL], 0, buf);
            break;
    }
}

main ()
{
    FORM * form;

    set_field_term (form, update_total); /* establish termination
                                           routine */
}
```

Function **set_field_buffer()** sets the column total field to the value **total** stored in **buf**. See the section above, “Setting and Reading Field Buffers,” for details on **field_buffer()** and **set_field_buffer()**.

For another example, consider Figure 10-43. It shows a common use for field initialization and termination—highlighting a field when it becomes current and removing the highlight when it is no longer current.

Figure 10-43 Field Initialization and Termination to Highlight Current Field

```
#include <form.h>

void bold_off (form)
FORM * form;
{
    /* remove highlight */

    set_field_back (current_field (form), A_UNDERLINE);
}

void bold_on (form)
FORM * form;
{
    /* highlight field */

    set_field_back (current_field (form), A_STANDOUT | A_UNDERLINE);
}

main ()
{
    FORM * form;

    /* establish initialization and termination routines */

    set_field_init (form, bold_on);
    set_field_term (form, bold_off);
}
```

If functions **set_form_init()**, **set_form_term()**, **set_field_init()**, or **set_field_term()** encounter an error, they return the following:

E_SYSTEM_ERROR - system error

As usual, if you want a specific default initialization or termination function for all forms or all fields, you can pass the appropriate set function a NULL form pointer. Passing a NULL form pointer to the access functions returns the current ETI default.

Manipulating the Current Field

The current field is the field where your end-user is positioned on the display screen. It changes as the end-user moves about the form entering or changing data. The cursor rests on the current field. To have your application program set or determine the current field, you use the following functions.

SYNTAX

```
int set_current_field (form, field)
FORM * form;
FIELD * field;

FIELD * current_field (form)
FORM * form;

int field_index (field)
FIELD * field;
```

The function **set_current_field()** enables you to set the current field, while function **current_field()** returns the pointer to it. The value returned by **field_index()** is the index to the given field in the field pointer array associated with the connected form. This value is in the range of 0 through N-1, where N is the total number of fields.

When a form is created by **new_form()** or the fields associated with the form are changed by **set_form_fields()** the current field is automatically set to the first visible, active field on page 0.

Note

Your application program need not call **set_current_field()** unless you want to implement field navigation requests that are not supported by the form driver and discussed in the earlier section, "ETI Form Requests."

Figure 10-44 illustrates the use of these functions. Function **set_first_field()** uses **set_current_field()** to set the current field to the first field in the form's field pointer array. Function **first_field()**, on the other hand, returns a Boolean value indicating whether the current field is the first field.

Figure 10-44 Example Manipulating the Current Field

```

int set_first_field (form) /* set current field to first field */
FORM * form;
{
    FIELD ** f = form_fields (form);
    return set_current_field (form, f[0]);
}

int first_field (form) /* check if current field is first field */
FORM * form;
{
    FIELD * f = current_field (form);
    return field_index (f) == 0;
}

```

If function **set_current_field()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer or field not connected to form
E_BAD_STATE	- called from init/term routines
E_INVALID_FIELD	- current field is invalid on posted form
E_REQUEST_DENIED	- field not active or not visible

The function **current_field()** returns (**FIELD ***) 0 if given a **NULL** form pointer or there are no fields connected to the form.

The function **field_index()** returns -1 if its field pointer argument is **NULL** or the field is not connected to a form.

Changing the Form Page

Two form functions enable your application program to change to another page on the form or to determine the current page of the form.

SYNTAX

```

int set_form_page (form, page)
FORM * form;
int page;

int form_page (form)
FORM * form;

```

Upon execution of **set_form_page()**, the current field is set to the first field on the new page that is visible and active (visited during form driver processing). Variable **page** must be in the range of 0 through **N-1**, where **N** is the total number of pages. The function **form_page()** returns the page number of the page currently visible on the screen.

Form Driver Processing

When function **new_form()** creates a form or function **set_form_fields()** changes the fields associated with a form, the form page is automatically set to 0.

Note

Your application program need not call **set_form_page()** unless you want to implement page navigation requests that are not supported by the form driver and discussed in the earlier section, "ETI Form Requests."

Figure 10-45 illustrates the use of these functions. Function **set_first_page()** uses **set_form_page()** to change to the first page of the form, while function **first_page()** uses **form_page()** to return a Boolean value indicating whether the first page of the form is currently displayed. Note that the first page is numbered 0.

Figure 10-45 Example Changing and Checking the Form Page Number

```
int set_first_page (form) /* set to first form page */
FORM * form;
{
    return set_form_page (form, 0);
}

int first_page (form) /* check if on the first form page */
FORM * form;
{
    return form_page (form) == 0; /* return Boolean */
}
```

If function **set_form_page()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form or page out of range
E_BAD_STATE	- called from init/term routines
E_INVALID_FIELD	- current field is invalid on posted form

The function **form_page()** returns -1 if given a NULL form pointer or there are no fields connected to the form.

Positioning the Form Cursor

As with menu processing, some processing of user form requests may move the cursor from the location required for continued processing by the form driver. This function moves the cursor back to where it belongs.

```
int pos_form_cursor (form)
FORM * form;
```

You need call this function only if your application program changes the cursor position of the form window.

Figure 10-46 illustrates one use of this function. Function **printpage()** repositions the cursor after it prints the page number in the form window.

Figure 10-46 Repositioning the Cursor After Printing Page Number

```
#include <form.h>
#include <curses.h>

void printpage (form)
FORM * form;
{
    int      p = form_page (form) + 1;
    WINDOW * w = form_win (form);
    int      rows, cols;
    char      buf[80];

    box (w, 0, 0);          /* put border around form window */
    getmaxyx (w, rows, cols); /* fetch window size */
    sprintf (buf, " %d ", p); /* store next page number */

    wmove (w, (rows-1), ((cols-1)-strlen(buf))/2);
                                /* position cursor */
    waddstr (w, buf);          /* print page number */

    /* position the form cursor for continued form processing */
    pos_form_cursor (form);
}

main ()
{
    FORM * form;

    set_form_init (form, printpage);
}
```

Form Driver Processing

If `pos_form_cursor()` encounters an error, it returns one of the following:

- | | |
|-----------------------------|----------------------|
| <code>E_SYSTEM_ERROR</code> | - system error |
| <code>E_BAD_ARGUMENT</code> | - null form pointer |
| <code>E_NOT_POSTED</code> | - form is not posted |

Setting and Fetching the Form User Pointer

As it does for items, menus, and fields, ETI supplies a form user pointer for data such as titles, help messages, and the like. These functions enable you to set the pointer and return its referent.

SYNTAX

```
int set_form_userptr (form, userptr)
FORM * form;
char * userptr;

char * form_userptr (form)
FORM * form;
```

You can define a structure to be connected to the form using this pointer. By default, the form user pointer is NULL.

Figure 10-47 illustrates the use of these form user pointer functions to determine whether a given name matches a pattern name. Function **main()** uses **set_form_userptr()** to establish the pattern name, while **compare()** uses **form_userptr()** to fetch the pattern and do the comparison.

Figure 10-47 Pattern Match Example Using form User Pointer

```
#include <form.h>

#define match(a,b)    (strcmp (a, b) == 0)

int compare (form, name)
FORM * form;
char * name;
{
    char * s = form_userptr (form); /* fetch pattern string */
    return match (name, s);        /* return Boolean indicating
                                   match or not */
}

main ()
{
    FORM * form;
    char * form_name; /* initialize form_name to desired string */

    set_form_userptr (form, form_name);
    /* set user pointer to point to string */
}
```


Setting and Fetching the Form User Pointer

For more user pointer examples, see the previous sections on item, menu, and field user pointers and the sample programs at the end of this guide.

If successful, **set_form_userptr()** returns **E_OK**. If not, it returns the following:

E_SYSTEM_ERROR	- system error
-----------------------	----------------

As usual, you change the default by passing **set_form_userptr()** a **NULL** form pointer. So, to change the default user pointer to point to the string ********, you write

```
/* change default user pointer */  
set_form_userptr((form *) 0, "****");
```

Setting and Fetching Form Options

ETI provides form options regulating how specific user requests are handled. These functions enable you to set the options and read their settings.

SYNTAX

```
int set_form_opts (form, opts)
FORM * form;
OPTIONS opts;
```

```
OPTIONS form_opts (form)
FORM * form;
```

```
options:
    O_NL_OVERLOAD
    O_BS_OVERLOAD
```

Note that function **set_form_opts()** automatically turns off all form options not referenced in its second argument. By default, all options are on.

The effects of the options are as follows:

O_NL_OVERLOAD determines how a **REQ_NEW_LINE** request is processed. If **O_NL_OVERLOAD** is on, the request is overloaded. See the section above, "Field Editing Requests," for a description of overloading. If **O_NL_OVERLOAD** is off, the **REQ_NEW_LINE** request behavior depends on whether insert mode is on.

In insert mode, the **REQ_NEW_LINE** request first inserts a new line after the current line. It then moves the text on the current line starting at the cursor position to the beginning of the new line. The cursor is repositioned to the beginning of the new line.

In overlay mode, the **REQ_NEW_LINE** request erases all data from the cursor position to the end of the line. It then repositions the cursor at the beginning of the next line.

Setting and Fetching Form Options

O_BS_OVERFLOW determines how a **REQ_DEL_PREV** request is processed. If **O_BS_OVERFLOW** is on, the request is overloaded. See again the section above, "Field Editing Requests," for information on overloading. If **O_BS_OVERFLOW** is off, the **REQ_DEL_PREV** request depends on whether insert mode is on.

In insert mode, if the cursor is at the beginning of any line except the first and the text on the line will fit at the end of the previous line, the text is appended to the previous line and the current line is deleted. If not, the **REQ_DEL_PREV** request simply deletes the previous character, if there is one. If the cursor is at the first character of the field, the form driver simply returns **E_REQUEST_DENIED**.

In overlay mode, the **REQ_DEL_PREV** request simply deletes the previous character, if there is one.

Options are Boolean values, so you use Boolean operators to turn them on or off. For example, to turn off option **O_NL_OVERFLOW** of form **f0** and turn on the same option of form **f1**, you write

```
FORM * f0, * f1;

set_form_opts (f0, form_opts (f0) & ~O_NL_OVERFLOW); /* turn option off */
set_form_opts (f1, form_opts (f1) | O_NL_OVERFLOW); /* turn option on */
```

ETI provides two more functions to turn options on and off.

```
int form_opts_on (form, opts)
FORM * form;
OPTIONS opts;

int form_opts_off (form, opts)
FORM * form;
OPTIONS opts;
```

Unlike function **set_form_opts()**, these functions do not affect options unreferenced in their second argument.

Setting and Fetching Form Options

Another way to turn off option `O_NL_OVERLOAD` on form `f0` and turn it on on form `f1` is to write

```
FORM * f0, * f1;

form_opts_off (f0, O_NL_OVERLOAD); /* turn option off */
form_opts_on  (f1, O_NL_OVERLOAD); /* turn option on  */
```

If functions `set_form_opts()`, `form_opts_off()`, or `form_opts_on()` encounter an error, they return the following:

<code>E_SYSTEM_ERROR</code>	-system error
-----------------------------	---------------

To change the current system default from, say, `O_NL_OVERLOAD` to `not-O_NL_OVERLOAD` without affecting the `O_BS_OVERLOAD` option, you write

```
form_opts_off( (FORM *) 0, O_NL_OVERLOAD);
```

Creating and Manipulating Programmer-Defined Field Types

In addition to the wealth of field types that ETI automatically provides, ETI lets you create new field types from old ones. For most applications, you may not need them, but when you do, you will have them.

Building a Field Type from Two Other Field Types

One way to define a new field type is to create one from two existing field types. The function `link_fielddtype()` lets you do this.

SYNTAX

```
FIELDTYPE * link_fielddtype(type1,type2)
FIELDTYPE * type1;
FIELDTYPE * type2;
```

The constituent types may be system-defined or programmer-defined types. They may require additional arguments for the later call to `set_field_type()` and may be associated with validation functions or choice functions. Validation functions validate the value in the field, while choice functions enable the user to choose the next or previous value of the field type. See the sections below, “Creating a Field Type with Validation Functions” and “Supporting Next and Previous Choice Functions.”

If additional arguments are required for the later call to `set_field_type`, those of `type1` should precede those of `type2`. If there are validation or choice functions associated with the constituent types, the new type first executes the function associated with `type1`. If it is successful, it returns TRUE. If not, the new type executes the function associated with `type2`. Whatever it returns is the value returned by the new type.

Creating and Manipulating Programmer-Defined Field Types

As an example, the following code creates a new field type that accepts either a color keyword or an integer between 0 and 255, inclusive:

```
FIELD *f1;

extern char ** colors;

ENUM_OR_INT = link_fldtype (TYPE_ENUM, TYPE_INTEGER);
/* Constituent types are System types
   described in section "Setting the Field
   Type to Ensure Validation" */

set_fldtype (f1, ENUM_OR_INT, colors, FALSE, FALSE, 0, 0L, 255L);
/* create field of field type ENUM_OR_INT */
```

Once you have created the new field type, you can create fields of that type. The last statement here creates field **f1**, which accepts only values of type **ENUM_OR_INT**.

If an error occurs, **link_fldtype()** returns the following:

NULL -no available memory

Creating a Field Type with Validation Functions

Another way to create a new field type is by specifying

- a function that validates each character as it is entered into the field
- a function that validates the entire value entered into the field
- both

Function **new_fldtype()** returns your new field type given pointers to these validation functions.

```
SYNTAX

typedef int (* PTF_int) ();

FIELDTYPE * new_fldtype (f_check, c_check)
PTF_int f_check;
PTF_int c_check;
```

The form driver automatically calls the named validation functions during form driver processing.

Creating and Manipulating Programmer-Defined Field Types

To create a new field type, you must write at least one of the two validation functions. Function **f_check** is a pointer to a function that takes two arguments: a field pointer and an argument pointer. The argument pointer is treated in the next section. **f_check** is called whenever the end-user tries to leave the field. It should check the field value stored in field buffer 0 and return TRUE if the field is valid or FALSE if not. If the validation function fails, your end-user remains on the offending field.

Function **c_check** is also a pointer to a function that takes two arguments: an integer that represents an ASCII character and an argument pointer. Function **c_check** is called as each character is entered by your end-user. It should check the character for validity and return TRUE if it is and FALSE if not.

Function **new_fieldtype()** is useful for creating field types for specialized applications. For example, Figure 10-48 defines a new field type **TYPE_HEX** as a hex number between **0x0000** and **0xffff**.

Creating and Manipulating Programmer-Defined Field Types

Figure 10-48 Creating a Programmer-Defined Field Type

```
#include <ctype.h>
#include <form.h>
extern long strtol ();

#define isblank(c) ((c) == ' ')

static int padding = 4; /* pad on left to 4 digits */
static long vmin = 0x0000L; /* minimum acceptable value */
static long vmax = 0xffffL; /* maximum acceptable value */

static int fcheck_hex (f, arg)
FIELD * f;
char * arg; /* unnecessary here, discussed in the next section */
{
    char buf[80];
    char * x = field_buffer (f, 0);
    while (*x && isblank (*x)) ++x;

    if (*x)
    {
        char * t = x;
        while (*x && isxdigit (*x)) ++x;
        while (*x && isblank (*x)) ++x;

        if (! *x)
        {
            long v = strtol (t, (char **) 0, 16);

            if (v >= vmin && v <= vmax)
            {
                sprintf (buf, "%.1lx", padding, v);
                set_field_buffer (f, 0, buf);
                return TRUE;
            }
        }
    }
    return FALSE;
}

static int ccheck_hex (c, arg)
int c;
char * arg; /* unnecessary in this example, discussed in the
next section */
{
    return isxdigit (c);
}

FIELDTYPE * TYPE_HEX = new fieldtype (fcheck_hex, ccheck_hex);
/* create new field type */
```

Creating and Manipulating Programmer-Defined Field Types

Later, you assign fields with the field type `TYPE_HEX` as you do with any field type and field:

```
FIELD * field;

set_field_type (field, TYPE_HEX);
```

Function `ccheck_hex()` checks that the input character is a valid hexadecimal digit, while function `fcheck_hex()` examines the field value for valid characters and checks the range. If successful, `fcheck_hex()` pads the field to four digits and returns `TRUE`. If not, it returns `FALSE`.

Note

The argument `arg` to functions `f_check` and `c_check` is not used in this version of the `TYPE_HEX` example because the new type does not require additional arguments to the `set_field_type()` routine.

If successful, `new_fieldtype()` returns a pointer to the new field type. If either argument to `new_fieldtype()` is a `NULL` pointer, the corresponding validation is not performed. If no memory is available or both function pointers are `NULL`, `new_fieldtype()` returns `NULL`.

Freeing Programmer-Defined Field Types

This function frees any space allocated for a field type created with `new_fieldtype()` or `link_fieldtype()`. Its argument is a field type pointer previously obtained from one of these functions.

```
SYNTAX

int free_fieldtype (fieldtype)
FIELDTYPE * fieldtype;
```


Creating and Manipulating Programmer-Defined Field Types

You may want to free the field type `TYPE_HEX` from the previous example once fields of that type have been processed. To do so, you write

```
/* create field type TYPE_HEX */
create fields of this type
free fields of this type */

free_fielddtype(TYPE_HEX); /* free programmer-defined type */
```

If successful, function `free_fielddtype()` returns `E_OK`. If an error occurs, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field type
<code>E_CONNECTED</code>	- type is connected to one or more fields

Once a field type is freed, you must not use it again. If you do, the effect is undefined.

Supporting Programmer-Defined Field Types

You may want to support some programmer-defined field types with additional arguments or with previous and next choice functions. This section explains how to do so.

Argument Support for Field Types

Some field types may require additional arguments to the `set_field_type()` routine, which sets the field type of a field. Function `set_fielddtype_arg()` takes as arguments pointers to functions that manage storage for the additional arguments.

SYNTAX

```
typedef char * (* PTF_charP) ();
typedef void (* PTF_void) ();

int set_fielddtype_arg (fielddtype, make_arg, copy_arg, free_arg)
FIELDTYPE * fielddtype;
PTF_charP make_arg;
PTF_charP copy_arg;
PTF_void free_arg;
```

Creating and Manipulating Programmer-Defined Field Types

You must write the functions referenced by pointers **make_arg**, **copy_arg**, and **free_arg**. These functions should do the following:

make_arg	allocate a structure for the field specific parameters to set_field_type() and return a pointer to the saved data
copy_arg	duplicate the structure created by make_arg
free_arg	free any storage allocated by make_arg or copy_arg

Function **make_arg** is called automatically when your application program calls **set_field_type()**. It takes one argument, a **va_list ***. (See **varargs(M)** for details.) Function **make_arg** in turn should call **va_arg()** for each additional argument to **set_field_type()** associated with the field type. Note that function **va_start()** is called by **set_field_type()** before **make_arg** gains control, while function **va_end()** is called by **set_field_type()** after **make_arg** returns.

Function **make_arg** must allocate space for the information associated with the additional arguments, save the information, and return the pointer to the information cast to a character pointer. It is this character pointer that is the argument **arg** to the other functions associated with the field type, namely **copy_arg**, **free_arg**, **f_check**, **c_check**, **next_choice**, and **prev_choice**.

Function **copy_arg** takes as its sole argument a pointer to existing argument information. It returns a pointer to a copy of this information. Function **free_arg()** takes as its sole argument a pointer to existing argument information. It should free any space allocated by **make_arg**.

Figure 10-49 illustrates how you can add padding and range arguments to our **TYPE_HEX** defined above.

Creating and Manipulating Programmer-Defined Field Types

Figure 10-49 Creating TYPE_HEX with Padding and Range Arguments

```
/* TYPE_HEX
   set_field_type (f, TYPE_HEX, padding, vmin, vmax);

   int padding;      for padding with leading zeros
   long vmin;        minimum acceptable value
   long vmax;        maximum acceptable value */

#include <form.h>
#include <ctype.h>
#include <varargs.h>
extern long strtol ();

#define isblank(c) ((c) == ' ')

typedef struct {
    int    padding;
    long   vmin, vmax;
} HEX;

static char * make_hex (ap)
va_list * ap;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));

    if (n)
    {
        n->padding = va_arg (*ap, int);
        n->vmin = va_arg (*ap, long);
        n->vmax = va_arg (*ap, long);
    }
    return (char *) n;
}

static char * copy_hex (arg)
char * arg;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));
    if (n) *n = *((HEX *) arg);
    return (char *) n;
}

static void free_hex (arg)
char * arg;
{
    free (arg);
}
```

(Continued on next page.)

Creating and Manipulating Programmer-Defined Field Types

Figure 10-49 Creating TYPE_HEX with Padding and Range Arguments
(Continued)

```
static int fcheck_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    int padding = n -> padding;
    long vmin = n -> vmin;
    long vmax = n -> vmax;
    char buf[80];
    char * x = field_buffer (f, 0);

    while (*x && isblank (*x)) ++x;

    if (*x)
    {
        char * t = x;

        while (*x && isxdigit (*x)) ++x;
        while (*x && isblank (*x)) ++x;

        if (! *x)
        {
            long v = strtol (t, (char **) 0, 16);

            if (v >= vmin && v <= vmax)
            {
                sprintf (buf, "%.1x", padding, v);
                set_field_buffer (f, 0, buf);
                return TRUE;
            }
        }
    }
    return FALSE;
}

static int ccheck_hex (c, arg)
int c;
char * arg;
{
    return isxdigit (c);
}

FIELDTYPE * TYPE_HEX = new_fldtype (fcheck_hex, ccheck_hex);
set_fldtype_arg (TYPE_HEX, make_hex, copy_hex, free_hex);
```

Creating and Manipulating Programmer-Defined Field Types

Later, to create a field that stores a hex number between 0x0000 and 0xffff, we have

```
set_field_type (field, TYPE_HEX, 4, 0x0000L, 0xffffL);
```

From this example, note that

- Your function **make_arg** (here, **make_hex()**) picks off the additional arguments to **set_field_type()** using **va_arg()**.
- Function **make_hex()** allocates a HEX structure, saves the information provided by the additional arguments, and returns a pointer to the saved information.
- Function **copy_hex()** allocates and copies a HEX structure.
- Function **free_hex()** frees a HEX structure.
- Functions **make_hex()** and **copy_hex()** return NULL if the memory allocation fails.
- Function **check_hex()** uses the argument information to do the necessary padding and range check and returns TRUE if successful.
- ETI's internal caller to **make_hex()** and **copy_hex()** automatically checks that the values (**arg**) returned from the functions are not NULL. Therefore, there is no need for functions (such as **fcheck_hex()**) that use these values to check that they are not NULL.

If successful, function **set_fieldtype_arg()** returns E_OK. If an error occurs, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- field type, make_arg copy_arg, or free_arg is NULL

Creating and Manipulating Programmer-Defined Field Types

Supporting Next and Previous Choice Functions

Some field types comprise a set of values from which your user chooses (enters) one. The following functions support those types that have a set of choices.

SYNTAX

```
typedef char * (* PTF_charP) ();

int set_fielddtype_choice (type, next_choice, prev_choice)
FIELDTYPE * type;
PTF_int      next_choice;
PTF_int      prev_choice;

int next_choice(f, arg);
FIELD * f;
char * arg;

int prev_choice(f, arg);
FIELD * f;
char * arg;
```

These functions enable the ETI form driver to support the REQ_NEXT_CHOICE and REQ_PREV_CHOICE requests mentioned in the earlier section, “Form Driver Processing.”

To support these requests, your application-defined functions **next_choice** and **prev_choice** must

- take two arguments: a pointer to the current field and a pointer to the value **arg** that the **make_arg** function (such as **make_hex()** above) returned
- use function **field_buffer()** to read the current value
- call function **set_field_buffer()** with **buffer** argument 0 to set the next or previous value
- return success or failure if there is no logically next or previous value

Both functions can be quite similar.

Figure 10-50 shows an implementation of function **next_choice()** for the field type TYPE_HEX as defined above, such that REQ_NEXT_CHOICE increments the current value and REQ_PREV_CHOICE decrements the current value.

Creating and Manipulating Programmer-Defined Field Types

Figure 10-50 Creating a Next Choice Function for a Field Type

```
static int next_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    long v = n -> vmin;
    char buf[80];
    char * x = field_buffer (f, 0);
    while (*x && isblank (*x)) ++x;
    if (*x)
    {
        v = strtol (x, (char **) 0, 16);
        if (v >= n -> vmin && v < n -> vmax)
            ++v;
    }
    sprintf (buf, "%.1lx", n -> padding, v);
    set_field_buffer (f, 0, buf);
    return TRUE;
}

static int prev_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    long v = n -> vmax;
    char buf[80];
    char * x = field_buffer (f, 0);
    while (*x && isblank (*x)) ++x;
    if (*x)
    {
        v = strtol (x, (char **) 0, 16);
        if (v > n -> vmin && v <= n -> vmax)
            --v;
    }
    sprintf (buf, "%.1lx", v -> padding, v);
    set_field_buffer (f, 0, buf);
    return TRUE;
}

/* associate previous and next choice functions */
set_fielddtype_choice (TYPE_HEX, next_hex, prev_hex);
```

If given a blank field, your functions **next_choice** and **prev_choice** should, of course, do something reasonable, such as setting the field to the first or last value of the type.

Creating and Manipulating Programmer-Defined Field Types

If function `set_fieldtype_choice()` encounters an error, it returns one of the following:

`E_SYSTEM_ERROR`

- system error

`E_BAD_ARGUMENT`

- either field type `next_choice`
or `prev_choice` is null

Other ETI Routines

Knowing how to use the basic ETI routines to get output and input and to work with windows, panels, menus, and forms, you can design screen management programs that meet the needs of many users. The ETI library, however, has routines that let you do still more in your program. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single ETI program.

You should be comfortable using the routines previously discussed and the other routines for I/O and window manipulation discussed on the **curses(S)** manual page before you try to use the following ETI features. The routines described under “Routines for Drawing Lines and Other Graphics” and “Routines for Using Soft Labels” are features that are new for UNIX System V Release 3.0 and later releases. If a program uses any of these routines, it may not run on earlier releases of the UNIX System. You must use the Release 3.0 version of the low-level ETI library on UNIX System V Release 3.0 to work with these routines.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in ETI programs. ETI uses the same names for glyphs as the VT100 line-drawing character set.

To use the alternate character set in an ETI program, you pass a set of variables whose names begin with `ACS_` to the ETI routine `waddch()` or a related routine. For example, `ACS_ULCORNER` is the variable for the upper left corner glyph. If a terminal has a line-drawing character for this glyph, `ACS_ULCORNER`'s value is the terminal's character for that glyph OR'ed (|) with the bit-mask `A_ALTCHARSET`. If no line-drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for `ACS_HLINE`, a horizontal line, is a - (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard `ACS_` names and their defaults are listed on the `curses(S)` manual page.

Part of an example program that uses line-drawing characters follows. The example uses the ETI routine `box()` to draw a box around a menu on a screen. `box()` uses the line-drawing characters by default or when | (the pipe) and - are chosen. (See `curses(S)`.) Up and down more indicators are drawn on the box border (using `ACS_UARROW` and `ACS_DARROW`) if the menu contained within the box continues above or below the screen:

Routines for Drawing Lines and Other Graphics

```
box(menuwin, ACS_VLINE, ACS_HLINE);
...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
    waddch(menuwin, ACS_UARROW);
else
    addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
    waddch(menuwin, ACS_DARROW);
else
    waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
    ACS_DARROW = 'V';
```

For more information, see `curses(S)` in the *Programmer's Reference*.

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The ETI library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for an ETI program to make use of them.

Let's briefly discuss most of the ETI routines needed to use soft labels: **slk_init()**, **slk_set()**, **slk_refresh()** and **slk_noutrefresh()**, **slk_clear()**, **slk_restore()**, **slk_attron()**, **slk_attrset()**, and **slk_attroff()**.

When you use soft labels in an ETI program, you have to call the routine **slk_init()** before **initscr()**. This sets an internal flag for **initscr()** to look at. The flag says to use the soft labels. If **initscr()** discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The ETI routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set()** takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh()** is comparable to **wnoutrefresh()** in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh()** commonly follows, **slk_noutrefresh()** is the function that is most commonly used to output the labels.

Just as **wrefresh()** is equivalent to a **wnoutrefresh()** followed by a **doupdate()**, so too the function **slk_refresh()** is equivalent to a **slk_noutrefresh()** followed by a **doupdate()**.

If **initscr()** removes the bottom line of **stdscr** to simulate soft labels, the routines **slk_attron()**, **slk_attrset()**, and **slk_atroff()** can be used to manipulate the appearance of the simulated soft labels. Note that these routines will have no effect on soft function key labels supplied by the terminal. These routines are similar to **attron()**, **attrset()**, and **attroff()** (see the section "Simple Input and Output" in this chapter).

To prevent the soft labels from getting in the way of a shell escape, **slk_clear()** may be called before doing the **endwin()**. This clears the soft labels off the screen and does a **doupdate()**. The function **slk_restore()** may be used to restore them to the screen. See the **curses(S)** manual page for more information about the routines for using soft labels.

Working with More than One Terminal

An ETI program can produce output on more than one terminal at the same time. This is useful for single-process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the ETI library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **\$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But for some applications, such as an inter-terminal communication program or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

An ETI program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary ETI routines.

References to terminals in an ETI program have the type **SCREEN***. A new terminal is initialized by calling **newterm(type, outfd, infd)**. **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio(S)** file pointer (**FILE***) used for output to the terminal, and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr()**, which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call `set_term(sp)` where *sp* is the screen reference to be made current. `set_term()` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm()`. Options such as `cbreak()` and `noecho()` must be set separately for each terminal. The functions `endwin()` and `refresh()` must be called separately for each terminal. Figure 10-51 shows a typical scenario to output a message to several terminals.

Figure 10-51 Sending a Message to Several Terminals

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Working with terminfo Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines because the higher level **curses** routines make your program more portable to other UNIX Systems and to a wider class of terminals.

Note

You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 10-52.

Figure 10-52 Typical Framework of a **terminfo** Program

```
#include <curses.h>
#include <term.h>
...
    setupterm( (char*)0, 1, (int*)0 );
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm()** takes care of initialization. Passing this routine the values **(char*)0**, **1**, and **(int*)0** invokes reasonable defaults. If **setupterm()** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode()** performs functions similar to **endwin()** and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm()**. It can be output using the **terminfo** routines **putp()** or **tputs()**, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf(S)**, because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch()** and **getch()** are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo(F)**; see **curses(S)** for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling, and the guidelines for running, a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections “Compiling an ETI Program” and “Running an ETI Program” in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see “Program Examples”) that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;    /* Currently underlining */

main(argc, argv)
    int argc;
    char **argv;
{
    FILE *fd;
    int c, c2;
    int outch();

    if (argc > 2)
    {
        fprintf(stderr, "Usage: termhl [file]\n");
        exit(1);
    }

    if (argc == 2)
    {
        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
            perror(argv[1]);
            exit(2);
        }
    }
    else
    {
        fd = stdin;
    }
    setupterm((char*)0, 1, (int*)0);
```

(Continued on Next Page)

(Continued)

```
for (;;)
{
    c = getc(fd);
    if (c == EOF)
        break;
    if (c == '\\')
    {
        c2 = getc(fd);
        switch (c2)
        {
            case 'B':
                tputs(enter_bold_mode, 1, outch);
                continue;
            case 'U':
                tputs(enter_underline_mode, 1, outch);
                ulmode = 1;
                continue;
            case 'N':
                tputs(exit_attribute_mode, 1, outch);
                ulmode = 0;
                continue;
        }
        putchar(c);
        putchar(c2);
    }
    else
        putchar(c);
}
fclose(fd);
fflush(stdout);
resetterm();
exit(0);
}
```

(Continued on next page)

(Continued)

```
/*
 * This function is like putchar, but it checks
 * for underlining.
 */
putch(c)
    int c;
{
    outch(c);
    if (ulmode && underline_char)
    {
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}

/*
 * Outchar is a function version of putchar that
 * can be passed to tputs as a routine to call.
 */
outch(c)
    int c;
{
    putchar(c);
}
```

Let's discuss the use of the function **tputs**(*cap*, *affcnt*, *outc*) in this program to gain some insight into the **terminfo** routines. **tputs**() applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like **\$<20>**, which means to pad for 20 milliseconds (see the section "Specify Capabilities" in this chapter). **tputs** generates enough pad characters to delay for the appropriate time.

tput() has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp**(*cap*) is a convenient abbreviation. **termhl** could be simplified by using **putp**()).

Now to understand why you should use the **curses**-level routines instead of **terminfo**-level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low-level details such as this are precisely why the **curses** level is recommended over the **terminfo** level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses(S)**) could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some UNIX System tools, like **vi(C)** (see the *User's Reference*), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier. (Lest we forget the UNIX System motto: Build on the work of others.)

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a -w) version of our fictitious terminal would be described as **myterm-w**. **term(M)** describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

Working with the terminfo Database

- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type

stty -echo; cat -vu

Type in the keys you want to test;

for example, see what right arrow (→) transmits.

<CR>

<CTRL-D>

stty echo

or

cat >dev/null

Type in the escape sequences you want to test;

for example, see what \E [H transmits.

<CTRL-D>

- The first line in each of these testing methods sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.
- See the **terminfo(F)** manual page. It lists all the capability names you have to use in a terminal description.

The following section, “Specify Capabilities,” gives details.

Specify Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal’s value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel=^G**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a # at the beginning of the line.

The **terminfo(F)** manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of

the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled “Capname.”

Note

For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal’s character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

- | | |
|----------|--|
| # | This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as cols#80 ,. |
| = | This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow: |
| ^ | This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as ^G . |
| \E or \e | These characters followed by another character show an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C . |
| \n | These characters provide a <NL> character sequence. |
| \l | These characters provide a linefeed character sequence. |
| \r | These characters provide a return character sequence. |
| \t | These characters provide a tab character sequence. |

Working with the terminfo Database

<code>\b</code>	These characters provide a backspace character sequence.
<code>\f</code>	These characters provide a formfeed character sequence.
<code>\s</code>	These characters provide a space character sequence.
<code>\nnn</code>	These are characters whose three-digit octal is <i>nnn</i> , where <i>nnn</i> can be one to three digits.
<code>\$< ></code>	These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the “less than/greater than” symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as <code>\$<20*></code> . See the terminfo(F) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel=^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of `myterm`. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description. The terminal capabilities follow:

- an automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**)
- the ability to produce a beeping sound [the instruction required to produce the beeping sound is **^G (bel)**].
- an 80-column wide screen (**cols**)
- a 30-line long screen (**lines**)
- use of xon/xoff protocol (**xon**)

By combining the name string (see the section “Name the Terminal”) and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
    am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal, **myterm**, has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A **<CR>** is a CTRL-M (**cr**).
- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cud1**).
- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse video mode is an ESCAPE-D (**sms0**).

Working with the terminfo Database

- Exiting reverse video mode is an ESCAPE-Z (**rmso**).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a <NL> at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,  
  am, bel=^G, cols#80, lines#30, xon,  
  cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,  
  smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[A (**kcuu1**).
- The down arrow key generates an ESCAPE-[B (**kcud1**).
- The right arrow key generates an ESCAPE-[C (**kcuf1**).
- The left arrow key generates an ESCAPE-[D (**kcub1**).
- The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,  
  am, bel=^G, cols#80, lines#30, xon,  
  cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,  
  smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0  
  kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,  
  kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters—for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm()** routine.

The arguments to string capabilities are manipulated with special ‘%’ sequences similar to those found in a **printf(S)** statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo(F)** for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal’s cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed by H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence ‘ESCAPE-[6 ; 19 H’ would be output.

Integer arguments are pushed onto the stack with a ‘%p’ sequence followed by the argument number, such as ‘%p2’ to push the second argument. A shorthand sequence to increment the first two arguments is ‘%i’. To output the top number on the stack as a decimal, a ‘%d’ sequence is used, exactly as in **printf**. Our terminal’s **cup** sequence is built up as follows:

cup=	Meaning
\E[output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E{%i%p1%d;%p2%dH,
```

Working with the terminfo Database

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,  
am, bel=^G, cols#80, lines#30, xon,  
cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,  
smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0  
kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,  
kcub1=\E[D, khome=\E[H,  
cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo(F)** for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named *myterm.ti*. The compiled description of myterm would usually be placed in */usr/lib/terminfo/m/myterm*, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to */f/fancy*. If the environment variable **\$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **\$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **\$TERMINFO** were set, before looking in the default */usr/lib/terminfo*. The general format for the **tic** compiler is as follows:

```
tic [-v] [-c] file
```

The **-v** option causes the compiler to trace its actions and output information about its progress. The **-c** option causes a check for errors; it may be combined with the **-v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat(C)** (see the *User's Reference*) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

```
tic -v myterm.ti<CR>  
(The trace information appears as the compilation  
proceeds.)
```

Refer to the **tic(ADM)** manual page in the *User's Reference* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **\$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing (using **vi(C)**—see the *User's Reference*) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput(C)** (see the *User's Reference*) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

```
tput [-Ttype] capname
```

The type of terminal you are requesting information about is identified with the **-Ttype** option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **\$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the “clear screen” character sequence for the terminal being used:

```
tput clear  
(The screen is cleared.)
```

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols  
(The number of columns used by the terminal appears here.)
```

The **tput(C)** manual page found in the *User's Reference* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(ADM) (see the *User's Reference*) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type

```
infocmp -I 5420
```

Converting a termcap Description to a terminfo Description

The **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo**(ADM) (see the *User's Reference*) command converts **termcap**(F) descriptions to **terminfo**(F) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example,

```
captoinfo /etc/termcap
```

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line

```
captoinfo
```

looks up the current terminal in the **termcap** database, as specified by the **\$TERM** and **\$TERMCAP** environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp -C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **-ltermcap** or **-ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **-lcurses** option.

TAM Transition Library

Character mode applications that run under the Terminal Access Method (TAM) can now run under ETI with a wide range of terminals. This section explains how to use the TAM transition library, the source of this portability. In addition, it explains how you can eventually rewrite your TAM application programs to run more efficiently under ETI without the TAM transition library.

Compiling and Running TAM Applications under ETI

The TAM transition library consists of a header file **tam.h** and a set of library routines. The file **tam.h** translates between TAM routines and equivalent sets of low-level ETI routines. For example, the TAM function **wcreate()** is mapped to the conversion library function **TAMwcreate()**, which consists of a series of low-level ETI calls, such as **newwin()** and **subwin()**.

To use the TAM transition library, be sure to include the standard TAM header file **tam.h** in your application program. So at the beginning of your TAM application program, you should already have

```
#include <tam.h> /* as usual, for TAM calls */
```

Next, you recompile and link your application program, say **tamprog.c**, to form an executable, as follows:

```
cc -I /usr/add-on/include tamprog.c -ltam -lcurses -o executable_name
```

Note the use of the **-I** option, which tells the compiler where to find the TAM header files. The two uses of the **-l** option link the requisite library subroutines, the TAM transition library and the low-level ETI library.

Alternatively, you might separately compile one or more TAM application files (say, **tam1.c**, **tam2.c**, and **main.c**) and later link them to form an executable program.

```
cc -c -I /usr/add-on/include tam1.c /* compile files individually */
cc -c -I /usr/add-on/include tam2.c

cc -c -I /usr/add-on/include main.c

/* link objects to form executable */
cc -o executable_name tam1.o tam2.o main.o -ltam -lcurses
```

Note that the **-I** option is required for the compilation of any file that uses the TAM library.

Tips for Polishing TAM Application Programs Running under ETI

To enable the code in your TAM application program to run smoothly under ETI, you should do the following:

- remove code that would be executed if a low-level **iswind()** function call returned a non-zero value, i.e., *true*. Under the TAM transition library, **iswind()** always returns *false*.
- remove all TAM calls to mouse management routines and the calls **wicon()**, **wicoff()**, and **wrastop()**, because they will translate to null operations.
- remove all machine-specific code, because the TAM transition library does not translate system calls specifically tailored to the UNIX System PC or calls [such as **ioctl(S)**] that have no meaning under ETI. These calls fail under the TAM transition library on all machines except the UNIX System PC.
- note that all calls to **track(S)** map to the low-level function **wgetc()**.
- remove all references to TAM calls that bear the same name as ETI calls because calls that have the same names in both systems have different effects.
- remove all arbitrary ANSI escape sequences for display output. For example, the TAM transition library does not recognize the escape sequence used on the UNIX System PC in the command **echo "\033[J"**, which clears the screen. Instead, you should use equivalent ETI routines (here, **clear()**).

Eliminating the superfluous code in the first three cases reduces your program's size and execution time.

How the TAM Transition Library Works

The TAM Transition Library translates between TAM function calls and low-level ETI function calls. It also ensures that escape and control sequences entered at a terminal's keyboard are properly interpreted.

Translations from TAM Calls to ETI Calls

The table in Figure 10-53 summarizes the translation of TAM to low-level ETI (**curses**) functions. Eventually, if you want to rewrite your TAM applications to make ETI calls directly and to run more efficiently, you can use this table as a guide.

Figure 10-53 Translations from TAM to ETI Function Calls
(Sheet 1 of 4)

TAM Function	Low-Level ETI (curses(S)) Equivalent
winit()	Calls initscr() .
wexit()	Calls endwin() and exit() .
iswind()	Returns FALSE.
wcreate()	Calls newwin() or new_panel() .
wdelete()	Calls delwin() or del_panel() .
wselect()	Calls touchwin() and wrefresh() , then updates the list of windows to indicate the new ordering.
wgetsel()	Calls top_panel() or bottom_panel() with NULL pointer.
wgetstat()	Calls getyx() , getmaxyx() , or getbegyx() .
wsetstat()	Calls del_panel() , then new_panel() .
wputc()	Calls waddch() .
wputs()	Calls waddstr() .
wprintf()	Calls wprintw() .
wslk()	Creates small window at bottom and uses curses routines with wprintw() .
wcmd()	Copies the character string passed by wcmd() to the bottom of the screen.

How the TAM Transition Library Works

Figure 10-53 Translations from TAM to ETI Function Calls
(Sheet 2 of 4)

TAM Function	Low-Level ETI (curses(S)) Equivalent
wprompt()	The character string passed by wprompt() is copied to the bottom of the screen.
wlabel()	The character string is printed in the upper left corner of the specified window.
wrefresh()	This calls wrefresh() . If the window index is -1, all windows should be refreshed in the appropriate order.
wuser()	This functionality is not necessary. Remove this from your code.
wgoto()	This calls wmove() .
wgetpos()	This calls getyx() .
wgetc()	This calls wgetch() . Character translation from ETI to ANSI may be required, depending on the current keypad mode.
kcodemap()	This functionality is not necessary. Remove this from your code.
keypad()	This calls keypad() .
wsetmouse()	This is a null operation.
wgetmouse()	This is a null operation.
wreadmouse()	This is a null operation.
wprexec()	This calls erase() and refresh() .
wpostwait()	This calls wrefresh() for each window in the window list.
wnl()	The functionality of this routine is not supported by curses .
wicon()	This is a null operation.
wicoff()	This is a null operation.
wrastop()	This is a null operation.
track()	This calls wgetch() .
initscr()	This calls initscr() .
nl()	The functionality of this routine is not supported by curses .
nonl()	The functionality of this routine is not supported by curses .

Figure 10-53 Translations from TAM to ETI Function Calls
(Sheet 3 of 4)

TAM Function	Low-Level ETI (curses(S)) Equivalent
<code>cbreak()</code>	Calls <code>cbreak()</code> .
<code>nocbreak()</code>	Calls <code>nocbreak()</code> .
<code>echo()</code>	Calls <code>echo()</code> .
<code>noecho()</code>	Calls <code>noecho()</code> .
<code>insch()</code>	Calls <code>insch()</code> .
<code>getch()</code>	Calls <code>getch()</code> .
<code>flushinp()</code>	Calls <code>flushinp()</code> .
<code>attron()</code>	Calls <code>attron()</code> .
<code>attroff()</code>	Calls <code>attroff()</code> .
<code>savetty()</code>	Calls <code>savetty()</code> .
<code>resetty()</code>	Calls <code>resetty()</code> .
<code>addch()</code>	Calls <code>addch()</code> .
<code>addstr()</code>	Calls <code>addstr()</code> .
<code>beep()</code>	Calls <code>beep()</code> .
<code>clear()</code>	Calls <code>clear()</code> .
<code>clearok()</code>	Null operation.
<code>clrtobot()</code>	Calls <code>clrtobot()</code> .
<code>clrtoeol()</code>	Calls <code>clrtoeol()</code> .
<code>delch()</code>	Calls <code>delch()</code> .
<code>deleteln()</code>	Calls <code>deleteln()</code> .
<code>erase()</code>	Calls <code>erase()</code> .
<code>flash()</code>	Calls <code>flash()</code> .
<code>getyx()</code>	Calls <code>wgetyx()</code> .
<code>insertln()</code>	Calls <code>insertln()</code> .
<code>leaveok()</code>	Null operation.
<code>move()</code>	Calls <code>move()</code> .
<code>mvaddch()</code>	Calls <code>move()</code> and <code>addch()</code> .
<code>mvaddstr()</code>	Calls <code>move()</code> and <code>addstr()</code> .
<code>mvinch()</code>	Calls <code>move()</code> and <code>inch()</code> .

How the TAM Transition Library Works

Figure 10-53 Translations from TAM to ETI Function Calls
(Sheet 4 of 4)

TAM Function	Low-Level ETI (curses(S)) Equivalent
nodelay()	Calls nodelay() .
wndelay()	Calls nodelay() .
refresh()	Calls refresh() .
resetterm()	Calls resetterm() .
baudrate()	Calls baudrate() .
endwin()	Calls endwin() .
fixterm()	Calls fixterm() .
printw()	Calls printw() .

Because the high-level TAM functions in the table in Figure 10-54 make calls only to the low-level functions in the previous table, you can continue to use those high-level TAM functions in your application programs as well. However, with ETI, you cannot use other TAM high-level functions such as **wtargeton()**.

Figure 10-54 TAM High-Level Functions

Usable TAM High Level Functions

form()	menu()	message()
pb_empty()	pb_gets()	adf_gttok()
pb_open()	pb_check()	pb_seek()
pb_name()	pb_puts()	pb_weof()
pb_gbuf()	adf_gtwrdr()	adf_gtxcd()
wind()	exhelp()	

The TAM Transition Keyboard Subsystem

Both TAM and ETI use a set of virtual function keys that translate between an escape character sequence entered at the keyboard and a bit pattern inside the machine. Under the TAM transition library, the TAM virtual key values are translated into ETI virtual key values.

The table in Figure 10-55 lists these equivalent virtual key values. Entering the escape sequence listed in the left column will generate the corresponding TAM virtual function key value given in the middle column. The right column lists the ETI equivalent of the TAM virtual key and is for reference only.

Figure 10-55 Translation Between TAM Escape Sequences and Virtual Key Values

TAM Escape Sequence	Virtual Key Value	
	TAM	ETI
ESC-!	s_F1	KEY_F(8)
ESC-@	s_F2	KEY_F(9)
ESC-#	s_F3	KEY_F(10)
ESC-\$	s_F4	KEY_F(11)
ESC-%	s_F5	KEY_F(12)
ESC-^	s_F6	KEY_F(13)
ESC-&	s_F7	KEY_F(14)
ESC-*	s_F8	KEY_F(15)
ESC-f1	PF1	KEY_F(16)
ESC-f2	PF2	KEY_F(17)
ESC-f3	PF3	KEY_F(18)
ESC-f4	PF4	KEY_F(19)
ESC-f5	PF5	KEY_F(20)
ESC-f6	PF6	KEY_F(21)
ESC-f7	PF7	KEY_F(22)
ESC-f8	PF8	KEY_F(23)
ESC-f9	PF9	KEY_F(24)
ESC-f0	PF10	KEY_F(25)
ESC-f-	PF11	KEY_F(26)
ESC-f=	PF12	KEY_F(27)
ESC-1	F1	KEY_F(0)
ESC-2	F2	KEY_F(C)
ESC-3	F3	KEY_F(S)
ESC-4	F4	KEY_F(S)
ESC-5	F5	KEY_F(F)
ESC-6	F6	KEY_F(M)
ESC-7	F7	KEY_F(6)
ESC-8	F8	KEY_F(7)
ESC-bg	Beg	KEY_BEG
ESC-BG	s_Beg	KEY_SBEG
ESC-br	Break	KEY_BREAK
ESC-bw	Back	KEY_LEFT
ESC-BW	s_Back	KEY_SLEFT
ESC-ce	Clear	KEY_CLEAR
ESC-CE	Clear	KEY_CLEAR
ESC-ci	ClearLine	KEY_EOL
ESC-CI	s_ClearLine	KEY_SEOL
ESC-cl	Close	KEY_CLOSE

How the TAM Transition Library Works

ESC-CL	Close	KEY_CLOSE
ESC-cm	Cmd	KEY_COMMAND
ESC-CM	s_Cmd	KEY_SCOMMAND
ESC-cn	Cancel	KEY_CANCEL
ESC-CN	s_Cancel	KEY_SCANCEL
ESC-cp	Copy	KEY_COPY
ESC-CP	s_Copy	KEY_SCOPY
ESC-cr	Creat	KEY_CREATE
ESC-CR	s_Creat	KEY_SCREATE
ESC-dc	DleteChar	KEY_DC
ESC-Del	DeleteChar	KEY_DC
ESC-DC	s_DeleteChar	KEY_SDC
ESC-dl	Dlete	KEY_DL
ESC-DL	s_Delete	KEY_SDL
ESC-dn	Down	KEY_DOWN
ESC-DN	RollDn	KEY_SF
ESC-en	End	KEY_END
ESC-EN	s_End	KEY_SEND
ESC-ESC	Esc	none
ESC-ex	Exit	KEY_EXIT
ESC-EX	s_Exit	KEY_SEXIT
ESC-fi	Find	KEY_FIND
ESC-FI	s_Find	KEY_SFIND
ESC-fw	Forward	KEY_RIGHT
ESC-FW	s_Forward	KEY_SRIGHT
ESC-hl	Help	KEY_HELP
ESC-?	Help	KEY_HELP
ESC-HL	s_Help	KEY_SHELP
ESC-hm	Home	KEY_HOME
ESC-HM	s_Home	KEY_SHOME
ESC-im	InputMode	KEY_IC
ESC-NJ	s_InputMode	KEY_SIC
ESC-mk	Mark	KEY_MARK
ESC-MK	Slect	KEY_SELECT
ESC-ms	Msg	KEY_MESSAGE
ESC-MS	s_Msg	KEY_SMESSAGE
ESC-mv	Move	KEY_MOVE
ESC-MV	s_Move	KEY_SMOVE
ESC-nx	Next	KEY_NEXT
ESC-NX	s_Next	KEY_SNEXT
ESC-op	Open	KEY_OPEN
ESC-OP	Close	KEY_CLOSE
ESC-ot	Opts	KEY_OPTIONS
ESC-OT	s_Opts	KEY_SOPTIONS

How the TAM Transition Library Works

ESC-pg	Page	KEY_NPAGE
ESC-PG	s_Page	KEY_PPAGE
ESC-pr	Print	KEY_PRINT
ESC-PR	s_Print	KEY_SPRINT
ESC-pv	Prev	KEY_PREVIOUS
ESC-PV	s_Prev	KEY_SPREVIOUS
ESC-rd	RollDn	KEY_SF
ESC-RD	RollDn	KEY_SF
ESC-re	Ref	KEY_REFERENCE
ESC-RE	Rstrt	KEY_RESTART
ESC-rf	Rfrsh	KEY_REFRESH
ESC-RF	Clear	KEY_CLEAR
ESC-rm	Rsume	KEY_RESUME
ESC-RM	s_Rsume	KEY_SRSUME
ESC-ro	Redo	KEY_REDO
ESC-RO	s_Redo	KEY_SREDO
ESC-rp	Rplac	KEY_REPLACE
ESC-RP	s_Rplac	KEY_SREPLACE
ESC-rs	Rstrt	KEY_REFERENCE
ESC-RS	Rstrt	KEY_RESTART
ESC-ru	RollUp	KEY_SR
ESC-RU	RollUp	KEY_SR
ESC-sl	Slect	KEY_SELECT
ESC-SL	Slect	KEY_SELECT
ESC-ss	Suspd	KEY_SUSPEND
ESC-SS	s_Suspd	KEY_SSUSPEND
ESC-sv	Save	KEY_SAVE
ESC-SV	s_Save	KEY_SSAVE
ESC-ud	Undo	KEY_UNDO
ESC-UD	s_Undo	KEY_SUNDO
ESC-up	Up	KEY_UP
ESC-UP	RollUp	KEY_SR

Some keyboards have one or more keys that emit escape sequences that are identical to the UNIX System PC keyboard sequences but do not match in terms of functionality. The function of an operationally incompatible key will always map to its **terminfo** specification. The TAM specific function implied by the same escape sequence will be accessible through the technique described above. Mechanisms in **curses(S)** automatically handle timing conflicts between actual keyboard function keys and UNIX System PC keyboard escape sequences.

Program Examples

The following programs demonstrate uses of low-level ETI (**curses**) functions. See the demonstration programs delivered on the ETI product diskettes for programs that use the high-level ETI functions.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move()**, **mvaddstr()**, **flash()**, **wnoutrefresh()** and **clrtoeol()** routines. These routines are all discussed in this chapter under “Working with **curses** Routines.”

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch()** routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch()**, **delch()**, **insertln()**, and **deleteln()** routines. These functions insert and delete a character or line. See **curses(S)** for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

Note

Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.

Program Examples

```
/* editor: A screen-oriented editor. The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <stdio.h>
#include <urses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
    extern void perror(), exit();
    int i, n, l;
    int c;
    int line = 0;
    FILE *fd;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s file\n", argv[0]);
        exit(1);
    }

    fd = fopen(argv[1], "r");
    if (fd == NULL)
    {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    cbreak();
    nonl();
    noecho();
    idlok(stdscr, TRUE);
    keypad(stdscr, TRUE);

    /* Read in the file */
    while ((c = getc(fd)) != EOF)
    {
        if (c == '\n')
            line++;
        if (line > LINES - 2)
            break;
        addch(c);
    }
    fclose(fd);
    move(0,0);
    refresh();
    edit();
}
```

```

/* Write out the file */
fd = fopen(argv[1], "w");
for (l = 0; l < LINES - 1; l++)
{
    n = len(l);
    for (i = 0; i < n; i++)
        putc(mvinch(l, i) & A_CHARTEXT, fd);
    putc('\n', fd);
}
fclose(fd);
endwin();
exit(0);
}

len(lineno)
int lineno;
{
    int linelen = COLS - 1;

    while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
        linelen--;
    return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
    int c;

    for (;;)
    {
        move(row, col);
        refresh();
        c = getch();

        /* Editor commands */
        switch (c)
        {

            /* hjkl and arrow keys: move cursor
             * in direction indicated */
            case 'h':
            case KEY_LEFT:
                if (col > 0)
                    col--;
                else
                    flash();
                break;

```

Program Examples

```
case 'j':
case KEY_DOWN:
    if (row < LINES - 1)
        row++;
    else
        flash();
    break;

case 'k':
case KEY_UP:
    if (row > 0)
        row--;
    else
        flash();
    break;

case 'l':
case KEY_RIGHT:
    if (col < COLS - 1)
        col++;
    else
        flash();
    break;
/* i: enter input mode */
case KEY_IC:
case 'i':
    input();
    break;

/* x: delete current character */
case KEY_DC:
case 'x':
    delch();
    break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
    move(++row, col = 0);
    insertln();
    input();
    break;

/* d: delete current line */
case KEY_DL:
case 'd':
    deleteln();
    break;

/* ^L: redraw screen */
case KEY_CLEAR:
case CTRL('L'):
    wrefresh(curscr);
    break;
```



```

        /* w: write and quit */
        case 'w':
            return;
        /* q: quit without writing */
        case 'q':
            endwin();
            exit(2);
        default:
            flash();
            break;
    }
}

/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
    int c;

    standout();
    mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;)
    {
        c = getch();
        if (c == CTRL('D') || c == KEY_EIC)
            break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES - 1, COLS - 20);
    clrtoeol();
    move(row, col);
    refresh();
}

```

The highlight Program

This program illustrates a use of the routine `attrset()`. **highlight** reads a text file and uses embedded escape sequences to control attributes. `\U` turns on underlining, `\B` turns on bold, and `\N` restores the default output attributes.

Note the first call to `scrollok()`, a routine that we have not previously discussed (see `curses(S)`). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, `scrollok()` automatically scrolls the terminal up a line and calls `refresh()`.

```

/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
    FILE *fd;
    int c, c2;
    void exit(), perror();

    if (argc != 2)
    {
        fprintf(stderr, "Usage: highlight file\n");
        exit(1);
    }

    fd = fopen(argv[1], "r");

    if (fd == NULL)
    {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);
    nonl();
    while ((c = getc(fd)) != EOF)
    {
        if (c == '\\')
        {
            c2 = getc(fd);
            switch (c2)
            {
                case 'B':
                    attrset(A_BOLD);
                    continue;
                case 'U':
                    attrset(A_UNDERLINE);
                    continue;
            }
        }
    }
}

```


Program Examples

```
        case 'N':
            attrset(0);
            continue;
        }
        addch(c);
        addch(c2);
    }
    else
        addch(c);
}
fclose(fd);
refresh();
endwin();
exit(0);
}
```

The scatter Program

This program takes the first **LINES - 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```

/*
 *   The scatter program.
 */

#include    <curses.h>
#include    <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS 160
char s[MAXLINES][MAXCOLS]; /* Screen Array */
int  T[MAXLINES][MAXCOLS]; /* Tag Array - Keeps      */
                                /* track of the number    */
                                /* of characters printed */
                                /* and their positions. */

main()
{
    register int row = 0,col = 0;
    register int c;
    int char_count = 0;
    time_t t;
    void exit(), srand();

    initscr();
    for(row = 0;row < MAXLINES;row++)
        for(col = 0;col < MAXCOLS;col++)
            s[row][col]=' ';

    col = row = 0;
    /* Read screen in */
    while ((c=getchar()) != EOF && row < LINES ) {

        if(c != '\n')
        {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')
                char_count++;
        }
        else
        {
            col = 0;
            row++;
        }
    }
}

```

Program Examples

```
time(&t); /* Seed the random number generator */
srand((unsigned)t);

while (char_count)
{
    row = rand() % LINES;
    col = (rand() >> 2) % COLS;
    if (T[row][col] != 1 && s[row][col] != ' ')
    {
        move(row, col);
        addch(s[row][col]);
        T[row][col] = 1;
        char_count--;
        refresh();
    }
}
endwin();
exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak()** so that you can depress the space bar without having to hit return; it calls **noecho()** to prevent the space from echoing on the screen. The **nonl()** routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok()** routine, which we also have not discussed, is called to allow insert and delete line. (See **curses(S)** for more information about these routines). Also notice that **clrtoeol()** and **clrtobot()** are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses()** program can be created. This type of input file is called a show script.


```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }

    if ((fd=fopen(argv[1], "r")) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for (line = 0; line < LINES; line++)
        {
            if (!fgets(linebuf, sizeof linebuf, fd))
            {
                clrtoebot();
                done();
            }
            move(line, 0);
            printw("%s", linebuf);
        }
        refresh();
        if (getch() == 'q')
            done();
    }
}
```

Program Examples

```
void done()
{
    move(LINES - 1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
    void done(), exit();
    unsigned sleep();
    char *getenv();
    int c;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
        exit(1);
    }
    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "wt");
    signal(SIGINT, done); /* die gracefully */
}
```

```

me = newterm(getenv("TERM"), stdout, stdin); /* initialize my tty */
you = newterm(argv[2], fdyou, fdyou); /* Initialize the other terminal */

set_term(me); /* Set modes for my terminal */
noecho(); /* turn off tty echo */
cbreak(); /* enter cbreak mode */
nonl(); /* Allow linefeed */
nodelay(stdscr, TRUE); /* No hang on input */

set_term(you); /* Set modes for other terminal */
noecho();
cbreak();
nonl();
nodelay(stdscr, TRUE);

/* Dump first screen full on my terminal */
dump_page(me);

/* Dump second screen full on the other terminal */
dump_page(you);

for (;;) /* for each screen full */
{
    set_term(me);
    c = getch();
    if (c == 'q') /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(me);

    set_term(you);
    c = getch();
    if (c == 'q') /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(you);
    sleep(1);
}
}
dump_page(term)
SCREEN *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line = 0; line < LINES - 1; line++) {
        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtoebot();
            done();
        }
        mvaddstr(line, 0, linebuf);
    }
}

```


Program Examples

```
    }
    stdout();
    mvprintw(LINES - 1, 0, "--More--");
    standend();
    refresh(); /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES - 1, 0); /* to lower left corner */

    clrtoeol(); /* clear bottom line */
    refresh(); /* flush out everything */
    endwin(); /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES - 1, 0); /* to lower left corner */
    clrtoeol(); /* clear bottom line */
    refresh(); /* flush out everything */
    endwin(); /* curses cleanup */
    exit(0);
}
```

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh()** for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh()** on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin()** routine (which we have not discussed — see **curses(S)**) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin()** for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()
{
    int i, c;
    char buf[120];
    void exit();

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0); /* top 3 lines */
    for (i = 0; i < LINES; i++)
        mvprintw(i, 0, "This is line %d of stdscr", i);

    for (;;)
    {
        refresh();
        c = getch();
        switch (c)
        {
            case 'c': /* Enter command from keyboard */
                werase(cmdwin);
                wprintw(cmdwin, "Enter command:");
                wmove(cmdwin, 2, 0);
                for (i = 0; i < COLS; i++)
                    waddch(cmdwin, '-');
                wmove(cmdwin, 1, 0);
                touchwin(cmdwin);
                wrefresh(cmdwin);
                wgetstr(cmdwin, buf);
                touchwin(stdscr);

                /*
                 * The command is now in buf.
                 * It should be processed here.
                 */

            case 'q':
                endwin();
                exit(0);
        }
    }
}
```

The colors Program

This program creates two windows. All characters displayed in the first window will be in red, on a blue background. All characters displayed in the second window will be in yellow, on a magenta background.

```
#include <curses.h>

#define PAIR1 1
#define PAIR2 2

main()
{
    WINDOW *win1, *win2;

    initscr();
    if ((start_color()) == OK)
    {
        /* create windows */
        win1 = newwin (5, 40, 0, 0);
        win2 = newwin (5, 40, 15, 40);

        /* create two color pairs */
        init_pair (PAIR1, COLOR_RED, COLOR_BLUE);
        init_pair (PAIR2, COLOR_YELLOW, COLOR_MAGENTA);

        /* turn on color attributes for each window */
        wattron (win1, COLOR_PAIR (PAIR1));
        wattron (win2, COLOR_PAIR (PAIR2));

        /* print some text in each window and exit */
        waddstr (win1, "This should be red on blue");
        waddstr (win2, "This should be yellow on magenta");
        wrefresh (win1);
        wrefresh (win2);

        /* wait for any key before terminating */
        wgetch (win2);
    }
    endwin();
}
```


Chapter 11

sdb: The Symbolic Debugger

Introduction 11-1

Using **sdb** 11-2

Printing a Stack Trace 11-3

Examining Variables 11-3

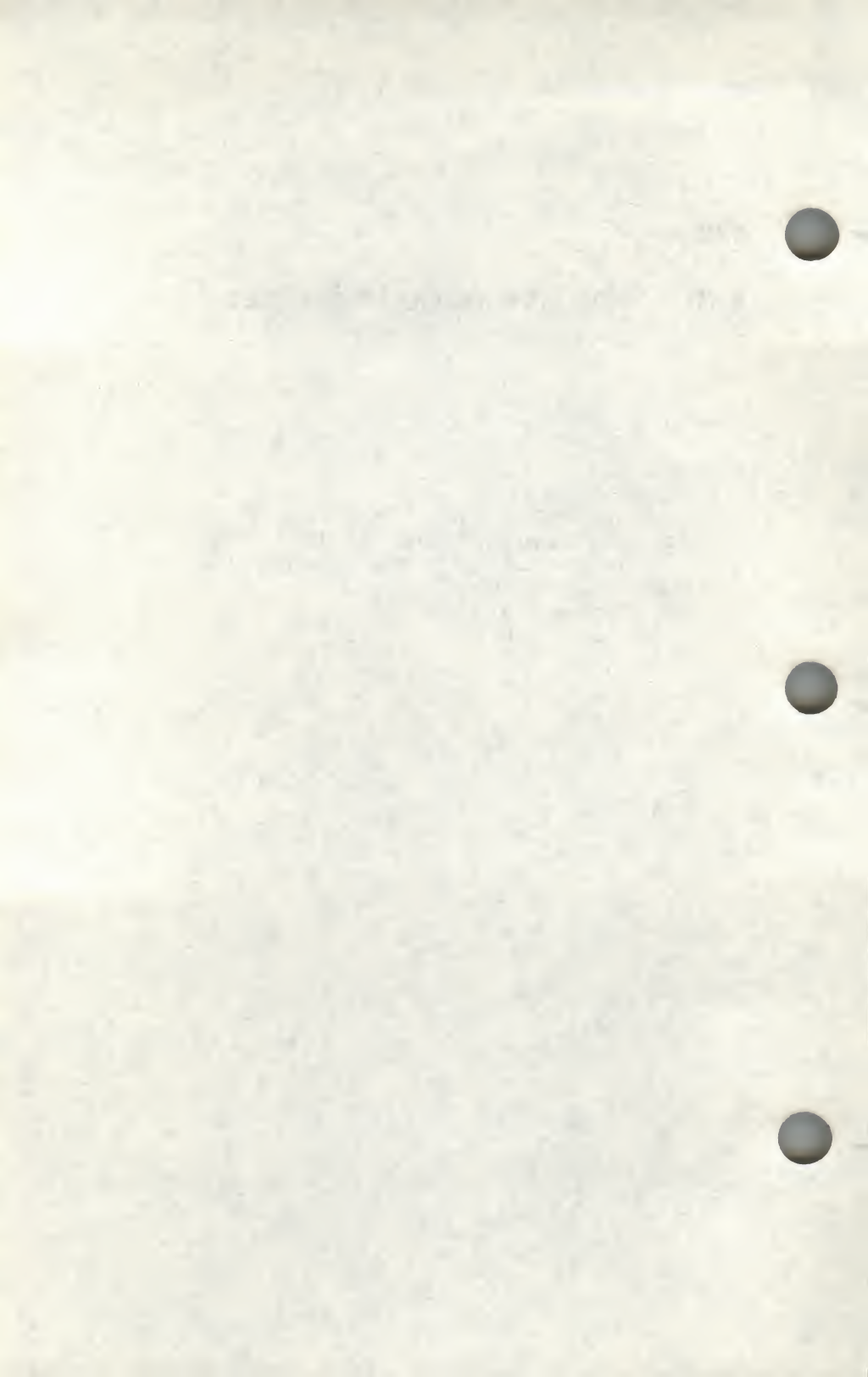
Source File Display and Manipulation 11-6

A Controlled Environment for Program Testing 11-8

Machine Language Debugging 11-11

Other Commands 11-13

An **sdb** Session 11-13



Introduction

This chapter describes the symbolic debugger, **sdb**(CP) in the *Programmer's Reference*, as implemented for C language programs on the UNIX System V/386 Release 3.2 Operating System. The **sdb** program is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source-language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and displayed in the correct format.

When executing, breakpoints may be placed at selected statements, or the program may be single-stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines, which give formatted printouts of structured data.

Using sdb

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of shell commands for debugging a core image is:

```
cc -g prgm.c -o prgm
prgm
Bus error - core dumped
sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred, causing a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *****, which shows that it is waiting for a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

Here **sdb** was called with one argument, **prgm**. In general, this takes three arguments on the command line:

1. The first argument is the name of the executable file that is to be debugged. It defaults to **a.out** when not specified.
2. The second argument is the name of the core file, defaulting to **core**.
3. The third argument is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory.

In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **-g** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **-g** option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the **-g** option.

Figure 11-1, at the end of the chapter, shows a more extensive example of **sdb** use.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls that led to the error. This is obtained with the **t** command. For example,

```
*t
sub(x=2,y=3)          [prgm.c:25]
inter(i=16012)        [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c) [prgm.c:15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file *prgm.c*. The **sub** function was called with the arguments *x*=2 and *y*=3 from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. A variable is displayed by typing its name followed by a slash, so:

```
*errflag/
```

causes **sdb** to display the value of variable **errflag**. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form:

```
*sub:i/
```

to display variable **i** in function **sub**.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol ***** is used to match any sequence of characters of a variable name and **?**, to match any single character. Consider the following commands:

```
*x*/  
*sub:y?/  
**/
```

The first prints the values of all variables beginning with **x**, the second prints the values of all two-letter variables in function **sub** beginning with **y**, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command:

```
**:* /
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, place a specifier after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

- b** one byte
- h** two bytes (half word)
- l** four bytes (long word)

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** format to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed.

There are a number of format specifiers available:

- c** character
- d** decimal
- u** decimal unsigned
- o** octal
- x** hexadecimal

- f** 32-bit single-precision floating point
- g** 64-bit double-precision floating point
- s** assume variable is a string pointer and print characters starting at the address pointed to by the variable, until a null is reached
- a** print characters starting at the variable's address until a null is reached
- p** pointer to function
- i** interpret as a machine-language instruction

For example, the variable **i** can be displayed with:

```
*i/x
```

which prints out the value of **i** in hexadecimal.

The **sdb** program also knows about structures, arrays, and pointers, so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Note that, as a special case,

```
*psym[0]
```

displays the structure pointed to by **psym** in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command:

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal, so the above command is equivalent to both:

```
*02000/
```

and:

```
*0x400/
```

Using sdb

It is possible to mix numbers and variables so that:

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and:

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the types `*1000.x/` and `*1000->x/`, the **sdb** program uses the structure template of the last structure referenced.

The address of a variable is printed with `=`, so:

```
*i=
```

displays the address of **i**. Another feature whose usefulness will become apparent later is the command:

```
*./
```

which redisplay the last variable typed.

Source File Display and Manipulation

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided that perform context searches within the source files of the program being debugged and display selected portions of the source files. The commands are similar to those of the UNIX System text editor **ed**(C), documented in the *User's Reference*. Like the editor, **sdb** has a notion of current file and line within the current file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and determining the context of the current line. The commands are as follows:

p	prints the current line
w	window; prints a window of ten lines around the current line
z	prints ten lines starting at the current line; advances the current line by ten
control-d	scrolls; prints the next ten lines and advances the current line by ten; used to display cleanly long segments of the program

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The **e** command is used to change the current source file. Either of the forms:

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

Using sdb

There are two commands for searching for instances of regular expressions in source files. They are:

```
*/regular expression/  
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression, and the second searches backwards. The trailing `/` and `?` may be omitted from these commands. Regular expression matching is identical to that of `ed(C)`.

The `+` and `-` commands may be used to move the current line forward or backward by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that:

```
*+15z
```

advances the current line by 15 and then prints ten lines.

A Controlled Environment for Program Testing

One very useful feature of `sdb` is breakpoint debugging. After entering `sdb`, breakpoints can be set at certain lines in the source program. The program is then started with an `sdb` command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and `sdb` reports the breakpoint where the program stopped. Now, `sdb` commands may be used to display the trace of function calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single-stepping. The `sdb` program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single-step through a function that has not been compiled with the `-g` option, execution proceeds until a statement in a function compiled with the `-g` option is reached. It is also possible to have the program execute one machine-level instruction at a time. This is particularly useful when the program has not been compiled with the `-g` option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function compiled with the **-g** option. The command format is as follows:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source-file display commands. The second form sets a breakpoint at line 12 of function **proc**, and the third sets a breakpoint at the first line of **proc**. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the **d** command:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or a **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command:

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the Program

11

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command:

```
*r args
```

runs the program with the given arguments as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases, after an appropriate message is printed, control returns to the user.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example,

```
*17 g
```

continues at line 17 of the current function. One use for this command is to avoid executing a section of code that is known to be bad. The user should not attempt to continue execution in a function different from that of the breakpoint.

The **s** command is used to run the program for a single statement. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when you are confident that the called function works correctly but are interested in testing the calling routine.

The **i** command is used to run the program, one machine-level instruction at a time, while ignoring the signal that stopped the program. Its uses are similar to the **s** command. There is also an **I** command that causes the program to execute one machine-level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print-structured data. There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...) /m
```

The first simply executes the function. The second is intended for calling functions. (It executes the function and prints the value that it returns.) The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

Machine Language Debugging

The **sdb** program has facilities for examining programs at the machine-language level. It is possible to print the machine-language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function **main**, use the command:

```
*main:25?
```

The **?** command is similar to the **/** command except that it searches backwards. The default format for printing text space is the **i** format, which interprets the machine-language instruction. The **control-d** command may be used to print the next ten instructions.

Using sdb

Absolute addresses may be specified instead of line numbers by appending a `:` to them, so that:

```
*0x1024:?
```

displays the contents of address 0x1024 in text space. Note that the command:

```
*0x1024?
```

displays the instruction corresponding to line 0x1024 in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address, so that:

```
*0x1024:b
```

sets a breakpoint at address 0x1024.

Manipulating Registers

The `x` command prints the values of all the registers. Also, individual registers may be named by appending a `%` sign to their name, so that on the 80286:

```
*ax%
```

displays the value of register `ax`, and on the 80386:

```
*eax%
```

displays the value of register `eax`.

Other Commands

To exit **sdb**, use the **q** command.

The **!** command (when used immediately after the ***** prompt) is identical to that in **ed(C)** and is used to have the shell execute a command. The **!** can also be used to change the values of variables or registers when the program is stopped at a breakpoint.

```
*variable!value
*eax!value
```

which sets the variable or the named register to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb Session

An example of a debugging session using **sdb** is shown in Figure 11-1. Comments (preceded by a pound sign, **#**) have been added to help you see what is happening.

Figure 11-1 Example of **sdb** Usage (Sheet 1 of 3)

```
sdb myoptim - ../common      # enter sdb command
Source path: ../common
No core image
*window:b                    # set a breakpoint at start of window
0x80802462 (window:1459+2) b
*r < m.s > out.m.s           # run the program
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean(*func)(); {
*t                            # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()      [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endif)    [local.c:483]
yylex()      [local.c:229]
main(argc=0,argv=0xc00201bc,-1073610300)    [optim.c:227]
```


Figure 11-1 Example of sdb Usage (Sheet 2 of 3)

```

11
#z                                # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:     extern NODE *initw();
1462:     register NODE *pl;
1463:     register int i;
1464:
1465:     TRACE(window);
1466:
1467:     /* find first window */
1468:
*s                                # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                # step
window:1465:     TRACE(window);
*s                                # step
window:1469:     wsize = size;
*s                                # step
window:1470:     if ((pl = initw(n0.forw)) == NULL)
*S                                # step through procedure call
window:1475:     for (opf = pf->back; ; opf = pf->back) {
*pl                               # show variable pl
0x80886b38
*x                                # print the 80286 register contents
    es/ 0x5f                      dx/ 0x67                      ds/ 0x5f
    ax/ 0x1cea                    cd/ 0x57                      ip/ 0x40
    di/ 0xfbd4                    cs/ 0x57                      bx/ 0xce6
    flgs/ 0x202                   bp/ 0x1cd6                     sp/ 0x1cce
    si/ 0x5b50                    ss/ 0x5f
0x570040 (main+4): lea -8(%bp),%d: [0x7fffffff]
*x                                # print the 80386 register contents
    eax/ 1                        ecx/ 0x402e2c                 edx/ 0xbffffca8
    eox/ 0x17                     esp/ 0xbffffc54                 edp/ 0xbffffc60
    esi/ 0x16                     edi/ 0x15                     eip/ 0x16b
    flag/ 0x13296                 trap/ 0xe                      err/ 7
0x16b (main+4):    lea    -8(%ebp),%edi    [0x7fffffff]

```

Figure 11-1 Example of sdb Usage (Sheet 3 of 3)

```

*pl[0]                                # dereference the pointer
pl[0].forw/ 0x80886b6c
pl[0].back/ 0x80886ac8
pl[0].ops[0]/ pushw
pl[0].unqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]                          # dereference the pointer
pl->forw[0].forw/ 0x80886ca0
pl->forw[0].back/ 0x80886b38
pl->forw[0].ops[0]/ call
pl->forw[0].unqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw                          # replace pl with pl->forw
*pl                                  # show pl
0x80886b6c
*c                                  # continue
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean (*func)(); {
*s                                  # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                  # step
window:1465: TRACE(window);
*size                              # show function argument size
3
*D                                  # delete all breakpoints
All breakpoints deleted
*c                                  # continue
Process terminated
*q                                  # quit sdb
$

```



Chapter 12

adb: A Program Debugger

Introduction 12-1

Starting and Stopping adb 12-2

Starting with a Program File 12-2

Starting with a Core Image File 12-3

Starting adb with Data Files 12-4

Starting with the Write Option 12-4

Starting with the Prompt Option 12-4

Leaving adb 12-5

Displaying Instructions and Data 12-6

Forming Addresses 12-6

Forming Expressions 12-7

Choosing Data Formats 12-12

Using the = Command 12-14

Using the ? and / Commands 12-15

An Example: Simple Formatting 12-16

Debugging Program Execution 12-18

Executing a Program 12-18

Setting Breakpoints 12-20

Displaying Breakpoints 12-21

Continuing Execution 12-21

Stopping a Program with Interrupt and Quit 12-21

Single-Stepping a Program 12-22

Killing a Program 12-22

Deleting Breakpoints 12-22

Displaying the C Stack Backtrace 12-23

Displaying CPU Registers 12-23

Displaying External Variables 12-24

A 286 Example: Tracing Multiple Functions 12-25

A 386 Example: Tracing Multiple Functions 12-30

Using the adb Memory Maps 12-35

Displaying the Memory Maps 12-35

Changing the Memory Map 12-38

Creating New Map Entries 12-39

Validating Addresses 12-40

Miscellaneous Features	12-41
Combining Commands on a Single Line	12-41
Creating adb Scripts	12-41
Setting Output Width	12-42
Setting the Maximum Offset	12-42
Setting Default Input Format	12-43
Using UNIX Commands	12-44
Computing Numbers and Displaying Text	12-44
An Example: Directory and Inode Dumps	12-45
 Patching Binary Files	12-48
Locating Values in a File	12-48
Writing to a File	12-49
Making Changes to Memory	12-49

Introduction

The **adb(CP)** program is a debugging tool for C and assembly language programs. It carefully controls the execution of a program while letting you examine and modify its data and text areas.

12

This chapter explains how to use **adb**. In particular, it explains how to:

- start the debugger
- display program instructions and data
- run, breakpoint, and single-step a program
- patch program files and memory

It also illustrates techniques for debugging C programs, and explains how to display information in non-ASCII data files.

Starting and Stopping adb

The **adb** program provides a set of commands that lets you examine, debug, and repair executable binary files as well as examine non-ASCII data files. To use these commands, you must invoke **adb** from a shell command line and specify the file or files you wish to debug. The following sections explain how to start **adb** and describe the types of files available for debugging.

Starting with a Program File

You can debug any executable C or assembly language program file using the following form:

```
adb [ filename ]
```

where *filename* is the name of the program file to be debugged. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the following command prepares the program named *sample* for examination and execution:

```
adb sample
```

Once started, **adb** prompts with an asterisk (*) and waits for you to enter commands. If you have given the name of a file that does not exist or is in the wrong format, **adb** will display an error message first, then wait for commands. For example, suppose you invoke **adb** with the following command:

```
adb sample
```

If the file *sample* does not exist, **adb** displays the following message:

```
adb: cannot open 'sample'
```

You can also start **adb** without a filename. In this case, **adb** searches for the default file *a.out* in your current working directory and prepares it for debugging. The *a.out* executable file is created by the C compiler when a program is compiled and linked successfully. Thus, typing:

```
adb
```

is the same as typing:

```
adb a.out
```

The **adb** program displays an error message and waits for a command if the *a.out* file does not exist.

12

Starting with a Core Image File

The **adb** program also lets you examine the core image files of programs that caused fatal system errors. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time the error occurred and provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the names of both the core and the program file. The command line has the following form:

```
adb programfile corefile
```

where:

- *programfile* is the filename of the program that caused the error, and
- *corefile* is the filename of the core image file generated by the system.

then **adb** uses information from both files to provide responses to your commands.

If you do not give a core image file, **adb** searches for the default *core* file in your current working directory. If such a file is found, **adb** uses it regardless of whether or not the file belongs to the given program. You can prevent **adb** from opening this file by using the hyphen (-) in place of the core filename. For example, the following command prevents **adb** from searching your current working directory for a core file:

```
adb sample -
```

Starting and Stopping **adb**

Starting **adb** with Data Files

You can use **adb** to examine a data file by giving the name of the data file in place of the program or core file. For example, to examine a data file named *outdata*, type:

```
adb outdata
```

The **adb** program opens this file and lets you examine its contents.

This method of examining files is very useful if the file contains non-ASCII data. The **adb** program provides a way to look at the contents of the file in a variety of formats and structures. Note that **adb** may display a warning when you give the name of a non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

Starting with the Write Option

You can make changes and corrections in a program or data file using **adb**, if you open it for writing using the **-w** option. For example, the following command opens the program file *sample* for writing:

```
adb -w sample
```

You can then use **adb** commands to examine and modify this file.

Note that the **-w** option causes **adb** to create a given file if it does not already exist. The option also lets you write directly to memory after executing the given program. See “Patching Binary Files.”

Starting with the Prompt Option

You can define your **adb** prompt using the **-p** option. The option has the following form:

```
-p prompt
```

where *prompt* is any combination of characters. If you use spaces, enclose the *prompt* in quotes. For example, the following command sets the prompt to Mar 10->

```
adb -p "Mar 10->" sample
```


The new prompt takes the place of the default prompt (*) when **adb** begins to prompt for commands.

Make sure there is at least one space between the **-p** and the new prompt; otherwise **adb** will display an error message. Note that **adb** automatically supplies a space at the end of the new prompt, so you need not.

Leaving adb

You can stop **adb** and return to the system shell using the **\$q** or **\$Q** command. You can also stop the debugger by pressing **<CTL>D**.

You cannot stop the **adb** command by pressing the Quit or Interrupt key. **adb** ignores Quit; Interrupt is caught by **adb** and causes it to wait for a new command.

Displaying Instructions and Data

The **adb** program provides several commands for displaying the instructions and data of a given program and the data of a given data file. The commands have the following form:

address [, *count*] = *format*

address [, *count*] ? *format*

address [, *count*] / *format*

where:

- *address* is a value or expression giving the location of the instruction or data item,
- *count* is an expression giving the number of items to be displayed, and
- *format* is an expression defining how to display the items.

The equal sign (=), question mark (?), and slash (/) tell **adb** from what source to take the item for display.

The following sections explain how to form addresses, how to choose formats, and the meaning of each of the display commands.

Forming Addresses

In **adb**, every address has the following form:

[*segment* :] *offset*

where:

- *segment* is an expression giving the address of a specific segment of 8086/286/386 memory, and
- *offset* is an expression giving an offset from the beginning of the specified segment to the desired item.

Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

The *segment*: is optional. If not given, the most recently typed segment is used.

12

Forming Expressions

Expressions contain decimal, octal, and hexadecimal integers, symbols, **adb** variables, register names, and a variety of arithmetic and logical operators.

Decimal, Octal, and Hexadecimal Integers

A decimal integer must begin with a nonzero decimal digit. An octal number must begin with a zero and may have octal digits only (union consists of 0-7). Hexadecimal numbers must begin with the prefix *0x* and may contain hexadecimal digits only, which consist of 0-9 and letters a-f in both upper and lowercase.

The following are valid numbers:

Decimal	Octal	Hexadecimal
34	042	0x22
4090	07772	0xffa

Although every decimal number is displayed with a trailing decimal point (.), you cannot use the decimal point when typing the number.

Symbols

A symbol is the name of a global variable or function defined within the program being debugged, and is equal to the address of the given variable or function. Symbols are stored in the program's symbol table, and are available if the symbol table has not been stripped from the program file. For more information, see **strip**(CP) in the *UNIX Programmer's Reference*.

Displaying Instructions and Data

When evaluating expressions that include functions, you can evaluate a function by specifying its name or its symbol table name. Symbols in the symbol table are no more than eight characters long, and those defined in C programs are given leading underscores (_). The following are examples of symbols:

```
main  _main hex2bin  __out_of
```

Note that if the spelling of any two symbols is the same, **adb** will ignore the second symbol and allow references only to the first. Also, using an underscore (_) in front of a name does not make a different symbol. For example, if both “main” and “_main” exist in a program, then **adb** accesses only the first to appear in the source and ignores the other.

When you use the question mark (?) command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the command sometimes gives a function name when it displays data. This does not happen if you use the ? command for text (instructions) and the slash (/) command for data. You cannot address local variables.

adb Variables

The **adb** program automatically creates a set of its own variables whenever you start the debugger. These variables are set to the addresses and sizes of various parts of the program file as defined below:

Variable	Definition
b	base address of data segment
d	size of data
e	entry address of the program
m	execution type
n	number of segments
s	size of stack
t	size of text

A user can access storage locations using the **adb** defined variables. The following request prints these variables:

```
$v
```

The **adb** program reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

You can use the current value of an **adb** variable in an expression by preceding the variable name with a less than (<) sign. For example, the current value of the base variable *b* is:

```
<b
```

You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater than (>) sign. The assignment has the following form:

```
expression > variable-name
```

where

- *expression* is the value to be assigned to the variable, and
- *variable-name* must be a single letter.

For example, the following assignment gives the hexadecimal value "0x2000" to the variable *b*:

```
0x2000>b
```

You can display the values of all currently defined **adb** variables using the **\$v** command. The command lists the variable names followed by their values in the current format. The command displays any variable whose value is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

Current Address

The **adb** program has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The dot (.) variable, also called the current address, contains the last address to be used in a command. The double quotation mark (") variable contains the last address to be typed with a command. The dot and double quote variables are usually the same except when you use

Displaying Instructions and Data

implied commands, such as the Newline and caret (^) characters. (These automatically increment and decrement dot, but leave " unchanged.)

You can use both the dot and the double quote in any expression. The less than (<) sign is not required. For example, the following command displays the value of the current address:

. =

and the following command displays the last address to be typed:

" =

Register Names

The **adb** program lets you use the current value of the CPU registers when evaluating expressions. You can give the value of a register by preceding its name with the less than (<) sign. The **adb** program recognizes the following register names:

286 Registers

ax	accumulator
cx	counter
dx	data
bx	base
sp	stack pointer
bp	base pointer
si	source index
di	destination index
es	extra segment
cs	code segment
ss	stack segment
ds	data segment
fl	flags register
ip	instruction pointer

386 Registers

eax	accumulator
ecx	counter
edx	data
ebx	base
esp	stack pointer
ebp	base pointer
esi	source index
edi	destination index
es	extra segment
cs	code segment
ss	stack segment
ds	data segment
fs	extra segment
gs	extra segment
efl	flags register
eip	instruction pointer

All 286 and 386 registers can be evaluated in expressions on XENIX 386, but only 286 registers can be evaluated in expressions on XENIX 286.

For example, the value of the 286 **ax** register can be evaluated in an expression by specifying the register as follows:

```
<ax
```

Note that you can not use register names unless either you start **adb** with a *core* file, or the program is currently being run under **adb** control.

12

Operators

You can combine integers, symbols, variables, and register names with the following operators:

Unary	Meaning
~	Not
-	Negative
*	Contents of location
Binary	Meaning
+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise And
	Bitwise inclusive Or
^	Modulo
#	Round up to the next multiple

Unary operators have higher precedence than binary operators. All binary operators have the same precedence. Thus, the following expression evaluates to 10:

```
2*3+4
```

and the following expression evaluates to 18:

```
4+2*3
```

You can change the precedence of the operations in an expression by using parentheses. For example, the following expression evaluates to 10:

```
4+(2*3)
```

Note

The **adb** program uses 32-bit arithmetic. This means that values that exceed 2,147,483,647 (decimal) are displayed as negative values.

The unary ***** operator treats an expression as a pointer to an address. An expression using this operator evaluates to the value stored at the given address. For example, the following expression evaluates to the value stored at the address

`“0x1234”:`

`*0x1234`

whereas the following is just equal to `“0x1234”`:

`0x1234`

Choosing Data Formats

Data of different forms can be displayed by specifying a string of format commands. A format command is a letter that specifies the format in which data is displayed. One or more letter commands can be concatenated with an integer to specify the number of times the letter commands are displayed.

The following illustrates each letter command and associated data format displayed:

Letter	Format
o	2 bytes in octal
d	2 bytes in decimal
D	4 bytes in decimal
x	2 bytes in hexadecimal
X	4 bytes in hexadecimal
u	2 bytes as an unsigned integer
f	4 bytes in floating point
F	8 bytes in floating point
c	1 byte as a character
s	A null terminated character string
i	Machine instruction
b	1 byte in octal
a	The current symbolic address
A	The current absolute address
n	A Newline
r	A blank space
t	A horizontal TAB

A letter command can be used by itself or combined with other commands to present a combination of data in different forms.

You can use the **d**, **o**, **x**, and **u** commands to display **int** type variables; you can use **D** and **X** to display **long** variables or 32-bit values. The **f** and **F** commands can be used to display single- and double-precision floating-point numbers. The **c** command displays **char** type variables, and the **s** command is for arrays of **char** that end with a null character (null terminated strings).

The **i** command displays machine instructions in 8086/286/386 mnemonics. The **b** command displays individual bytes and is useful for displaying data associated with instructions, or the high or low bytes of registers.

You usually combine the **a**, **r**, and **n** commands with other commands to make the display more readable. For example, the following format commands display the current address after each instruction:

ia

Displaying Instructions and Data

You can precede each format with a count of the number of times you wish it to be repeated. For example, the following format commands display four ASCII characters:

```
4c
```

You can also combine format requests to provide elaborate displays. For instance, the following commands display four octal words followed by their ASCII interpretation from the data space of the core image file:

```
<b, -1/4o4^8Cn
```

In this example, the display starts at the address “<b,” the base address of the program’s data. The display continues until the end-of-the-file since the negative count “-1” causes an indefinite execution of the commands until an error condition, such as the end of the file, occurs. The command **4o** displays the next four words (16-bit values) as octal numbers. The command **4^** then moves the current address back to the beginning of these four words and the **C** command redisplayes them as eight ASCII characters. Finally, **n** sends a Newline character to the terminal. The **C** command causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an at sign (@) followed by a lowercase letter. For example, the value 0 is displayed as @. The at sign itself is displayed as a double at sign (@@).

Using the = Command

The equal sign (=) command displays a given address in a given format. The command is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, typing the following displays the absolute address of the symbol “main” (giving the segment and offset):

```
main=A
```

Typing the following displays (in decimal) the sum of the variable *b* and the hexadecimal value *0x2000*:

```
<b+0x2000=D
```

If a count is given, the same value is repeated that number of times. For example, typing the following displays the value of “main” twice:

```
main,2=x
```

If no address is given, the current address is used instead. This is the same as the following command:

```
.=
```

If you do not specify a format, the previous format given for this command is used. For example, in the following sequence, both “main” and “_start” are displayed in hexadecimal:

```
main=x
_start=
```

12

Using the ? and / Commands

You can display the contents of a text or data segment with the ? and / commands. The commands have the following form:

```
[ address ] [, count ] ? [ format ]
```

```
[ address ] [, count ] / [ format ]
```

where:

- *address* is an address with the given segment,
- *count* is the number of items you wish to display, and
- *format* is the format of the items you wish to display.

You use the ? command to display instructions in the text segment. For example, the following command displays five instructions starting at the address “main,” and the address of each instruction displays immediately before it:

```
main,5?ia
```

The following command displays the instructions, with no addresses other than the starting address:

```
main,5?i
```

You use the / command to check the values of variables in a program, especially variables for which no name exists in the program’s symbol table. For example, the following command displays the value (in hexadecimal) of a local variable:

```
<bp-4?x
```

Displaying Instructions and Data

Local variables are generally at some offset from the address indicated by the **bp** register.

An Example: Simple Formatting

The following example illustrates how to combine formats in `%` or `/` commands, to display different types of values when stored together in the same program. This program has the following source statements:

```
char  str1[ ]      = "This is a character string" ;
int    one         = 1 ;
int    number      = 456 ;
long   lnum        = 1234 ;
float  fpt         = 1.25 ;
char  str2[ ]      = "This is the second character string" ;

main()
{
    one = 2;
}
```

The program is compiled and stored in a file named *sample*.

To start the session, type:

```
adb sample -
```

You can display the value of each individual variable by giving its name and corresponding format in a `/` command. For example, typing:

```
str1/s
```

displays the contents of *str1* as a string:

```
_str1:      This is a character string
```

The following command:

```
number/d
```

displays the contents of *number* as a decimal integer:

```
_number:    456.
```


You can choose to view a variable in a variety of formats. For example, you can display the **long** variable *lnum* as a 4-byte decimal, octal, and hexadecimal number by typing the following:

```
lnum/D
_lnum:      1234
lnum/O
_lnum:      02322
lnum/X
_lnum:      0x4d2
```

12

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, type:

```
str1,5/8x
```

This command displays eight hexadecimal values on a line, and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by typing:

```
str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times, immediately before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters, and give an address for each line by typing

```
str1,5/4x4^8t8Cna
```

Debugging Program Execution

The **adb** program provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Note that C compiler does not normally generate statement labels for programs. This means it is not possible to refer to individual C statements when using the debugger. In order to use execution commands effectively, you must be familiar with the instructions generated by the C compiler and how they relate to individual C statements. One useful technique is to create an assembly-language listing of your C program before using **adb**, then refer to the listing as you use the debugger. To create an assembly-language listing, use the **-S** option of the **cc** command. For more information, see **cc(CP)** in the *UNIX Programmer's Reference*.

Executing a Program

You can execute a program using the **:r** or **:R** command. The command has the following form:

```
[ address ] [,count ] :r [ arguments ]
```

```
[ address ] [,count ] :R [ arguments ]
```

where:

- *address* gives the address at which to start execution,
- *count* is the number of breakpoints you wish to skip before one is taken, and
- *arguments* are the command line arguments, such as filenames and options, that you wish to pass to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning, type:

```
:r
```

If a *count* is given, **adb** will ignore all breakpoints until the given number have been encountered. For example, the following command causes **adb** to skip the first 5 breakpoints:

```
,5:r
```

If you specify arguments, each of them must be separated by at least one space. The arguments are passed to the program in the same way the system shell passes command-line arguments to a program. You may use the shell-redirection symbols if you wish.

The **:R** command passes the command arguments through the shell before starting program execution. This means you can use shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The **:R** command is especially useful if the program expects multiple filenames. For example, the following command passes the argument "[a-z]*.s" to the shell where it is expanded to a list of the corresponding filenames before being passed to the program:

```
:R [a-z]*.s
```

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program you may have been running.

Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the following form:

12

```
address [, count ] :br command
```

where:

- *address* must be a valid instruction address,
- *count* is a count of the number of times you wish the breakpoint to be skipped before it causes the program to stop, and
- *command* is the **adb** command you wish to execute when the breakpoint is taken.

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. For example, the following command sets a breakpoint at the start of the function named “main”:

```
main:br
```

The breakpoint is taken just as control enters the function and before the function’s stack frame is created.

A breakpoint with a count is typically used within a function, that is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint lets the program continue to execute until the given function or instructions have been executed for the specified number of times. For example, the following command sets a breakpoint at the fifth repetition of the function “light”:

```
light,5:br
```

The breakpoint does not stop the function until it has been called at least five times.

Note that no more than 16 breakpoints at a time are allowed.

Displaying Breakpoints

You can display the location and count of each currently defined breakpoint by using the **\$b** command. The command displays a list of the breakpoints given by address. If the breakpoint has a count and/or a command, these are given as well.

The **\$b** command is useful if you have created several breakpoints in your program.

Continuing Execution

You can continue program execution after it has been stopped by a breakpoint by using the **:co** command. The command has the following form:

```
[ address ] [,count] :co [signal]
```

where:

- *address* is the address of the instruction at which you wish to continue execution,
- *count* is the number of breakpoints you wish to ignore, and
- *signal* is the number of the signal to send to the program. For more information, see **signal(S)** in the *UNIX Programmer's Reference*.

If you don't specify an *address*, the program starts at the next instruction after the breakpoint. If you do specify *count*, **adb** ignores the first *count* breakpoints.

Stopping a Program with Interrupt and Quit

You can stop program execution at any time by pressing the Interrupt, **<CTL>**, or Quit keys. These keys stop the current program and return control to **adb**. The keys are especially useful for programs that have infinite loops or other program errors.

Note that whenever you press the Interrupt, **<CTL>**, or Quit key to stop a program, **adb** automatically saves the signal and passes it to the program, if you start it again by using the **:co** command. This is very useful if you wish to test a program that uses these signals as part of its processing.

Debugging Program Execution

If you wish to continue program execution, but you do not wish to send the signals, type:

```
:co 0
```

The command argument **0** prevents a pending signal from being sent to the program.

12

Single-Stepping a Program

You can single-step a program, that is, execute it one instruction at a time, using the **:s** command. The command executes an instruction and returns control to **adb**. The command has the following form:

```
[address] [, count] :s
```

where:

- *address* must be the address of the instruction you wish to execute, and
- *count* is the number of commands you wish to execute.

If you do not specify an *address*, **adb** uses the current address. If you specify a *count*, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the following command executes the first 5 instructions in the function *main*:

```
main, 5:s
```

Killing a Program

You can kill the program you are debugging by using the **:k** command. The command kills the process created for the program and returns control to **adb**. The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

Deleting Breakpoints

You can delete a breakpoint from a program by using the **:dl** command. The command has the following form:

```
address :dl
```

where *address* is the address of the breakpoint you wish to delete.

The **:dl** command is typically used to delete breakpoints you no longer wish to use. Typing the following deletes the breakpoint set at the start of the function "main":

```
main:dl
```

Displaying the C Stack Backtrace

12

You can trace the path of all active functions by using the **\$c** command. The command lists the names of all functions that have been called but have not yet returned control, as well as the address from which each function was called, and the arguments passed to it.

For example, the following command displays a backtrace of the C language functions called:

```
$c
```

By default, the **\$c** command displays all calls. If you wish to display just a few, you must supply a count of the number of calls you wish to see. For example, the following command displays upto 25 calls in the current call path:

```
,25$c
```

Note that function calls and arguments are put on the stack after the function has been called. If you put breakpoints at the entry point to a function, the function will not appear in the list generated by the **\$c** command. You can remedy this problem by placing breakpoints a few instructions into the function.

Displaying CPU Registers

You can display the contents of all CPU registers by using the **\$r** command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter, and the instruction at the current address.

Debugging Program Execution

Registers for UNIX 286

The **adb** program displays registers in the following format when executing in UNIX 286; the value of each register is given in the current default format:

```
ax      0x0      fl      0x0
bx      0x0      ip      0x0
cx      0x0      cs      0x0
dx      0x0      ds      0x0
di      0x0      ss      0x0
si      0x0      es      0x0
sp      0x0      sp      0x0
0:0:    addb    al,bl
```

Registers for UNIX 386

The **adb** program displays registers in the following format when executing in UNIX 386; the value of each register is given in the current default format:

```
eax      0x81000    efl      0x246
ebx      0x0        eip      0x142
ecx      0x0        cs       0x3f
edx      0x8        ds       0x47
edi      0x0        es       0x47
esi      0x0        fs       0x47
ebp      0x0        gs       0x47
esp      0x7fef8    ss       0x47
0x3f:0x142:    push    ebp
```

Displaying External Variables

You can display the values of all external variables in a program by using the **\$e** command. External variables are variables in your program that have global scope, or have been defined outside of any function. This may include variables that have been defined in library routines used by your program.

The **\$e** command is useful whenever you need a list of the names for all available variables, or to quickly summarize their values. The command displays one name on each line with the variable's value (if any) on the same line.

For example, use the \$e command to display the following external variables and their values in hexadecimal format in a program:

```
__environ: 0xff08
__fcnt:    0x0
__gcnt:    0x0
__hcnt:    0x0
__errno:   0x0
__bufend:  0x0
__xcstar:  0x0
__xistar:  0x0
__end:     0x0
__stdbuf:  0x1b0
__iob:     0x0
__edata:   0x0
__argv:    0xff00
__acrtus:  0x0
__xcend:   0x0
__sibuf:   0x0
__lastbu:  0x130
__xiend:   0x0
__smbuf:   0x0
__sobuf:   0x0
```

A 286 Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:


```

int      fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}

```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br** command. For example, to set a breakpoint at the start of function "f," type:

```
f:br
```

You can use similar commands for the “g” and “h” functions. Once you have created the breakpoints, you can display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count bkpt      command
1      _h
1      _g
1      _f
```

The next step is to display the first five instructions in the “f” function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address.

```
_f:      push      bp
_f+1.:   mov       bp,sp
_f+3.:   mov       ax,4
_f+6.:   call      near __chkstk
_f+9.:   push      di
_f+10.:  
```

You can display five instructions in the “g” function without their addresses by typing:

```
g,5?i
```

The system displays:

```
_g:      push      bp
         mov       bp,sp
         mov       0x4
         call      near __chkstk
         push      di
```

To begin program execution, type:

```
:r
```

then **adb** displays the following message and begins to execute:

```
sample: running
```

Debugging Program Execution

As soon as **adb** encounters the first breakpoint (at the beginning of the “f” function), it stops execution and displays the following message:

```
breakpoint _f:  push bp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

You can continue the program by typing:

```
:co
```

The **adb** program displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint _g:  push bp
```

You can now trace the path of execution by typing:

```
$c
```

The display shows that only three functions are active: “f,” “main,” and “start”:

```
__f (1., 1.)                from __main+18.  
__main (1., 5922., 5926.)   from __start+50.  
__start                    from start0+5.
```

The values 18, 5922, and 5926 will vary.

Although the breakpoint has been set at the start of function “g,” it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

adb responds with a message notifying you that the execution has stopped after **adb** single-stepped the first five instructions. Now you can list the backtrace again. Type:

```
$c
```


This time, the list shows four active functions:

```

_g (2., 3.)           from _f+39
_f (1., 1.)          from _main+18
_main (1., 5922., 5926) from _start+50
_start ()            from start0+5

```

You can display the contents of the integer variable *fcnt* by typing:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be 1. You can continue execution of the program and skip the first 10 breakpoints by typing:

```
,10:co
```

In response to this, **adb** starts the program and then displays the running message again. The program does not stop until **adb** encounters exactly 10 breakpoints, when it displays the following message:

```
breakpoint _g: push bp
```

To show that these breakpoints have been skipped, you can display the backtrace again, by typing:

```
$c
```

For UNIX 286, your system displays:

```

_f (2., 11.)           from _h+36.
_h (10., 9.)          from _g+38.
_g (11., 20.)         from _f+39.
_f (2., 9.)           from _h+36.
_h (8., 7.)           from _g+38.
_g (9., 16.)          from _f+39.
_f (2., 7.)           from _h+36.
_h (6., 5.)           from _g+38.
_g (7., 12.)          from _f+39.
_f (2., 5.)           from _h+36.
_h (4., 3.)           from _g+38.
_g (5., 8.)           from _f+39.
_f (2., 3.)           from _h+36.
_h (2., 1.)           from _g+38.
_g (2., 3.)           from _f+39.
_f (1., 1.)           from _main+18.
_main (1., 5922., 5926.) from _start+50.
_start ()             from start0+5.

```

Exit **adb** by typing:

```
$q
```

A 386 Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:

```

int      fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}

```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the `:br` command. For example, to set a breakpoint at the start of function “f,” type:

```
f:br
```

You can use similar commands for the “g” and “h” functions. Once you have created the breakpoints, you can display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count bkpt      command
1      h
1      g
1      f
```

The next step is to display the first five instructions in the “f” function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address:

```
f:      push  ebp
f+0x1:  mov   ebp,esp
f+0x3:  sub   esp,0x8
f+0x9:  push  ebx
f+0xa:  push  edi
f+0xb:
```

You can display five instructions in the “g” function without their addresses by typing:

```
g,5?i
```

In this case, the display is:

```
g:      push  ebp
        mov   ebp,esp
        sub   esp,0x8
        push  ebx
        push  edi
```


Debugging Program Execution

To begin program execution, type:

```
:r
```

then **adb** displays the following message and begins to execute:

```
sample: running
```

As soon as **adb** encounters the first breakpoint (at the beginning of the “f” function), it stops execution and displays the following message:

```
breakpoint f:  push ebp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

You can continue the program by typing:

```
:co
```

then **adb** displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint g:  push ebp
```

You can now trace the path of execution by typing:

```
$c
```

The commands show that only three functions are active: “f,” “main,” and “start”:

```
  _f (0x1, 0x1)                                from _main+0x15  
  _main(0x1, 0x187ef20, 0x187ef28) from __start+0x39
```

The values 0x187ef20, 0x187ef28, and 0x39 will vary.

Although the breakpoint has been set at the start of function “g,” it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

The **adb** program responds with a message indicating it has single-stepped the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```

_g (0x2,0x3)           from _f+0x2c
_f (0x1,0x1)           from _main+0x15
_main (0x1, 0x187ef20, 0x187ef28) from __start+0x39

```

12

You can display the contents of the integer variable *fcnt* by typing:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be *1*. You can continue execution of the program and skip the first 10 breakpoints by typing:

```
,10:co
```

now **adb** starts the program; then it displays the running message again. It does not stop the program until it encounters exactly ten breakpoints. It displays the following message:

```
breakpoint g:  push ebp
```

To show that these breakpoints have been skipped, you can display the backtrace again, by typing:

```
$c
```

Debugging Program Execution

For UNIX 386, your system displays:

```

_f (0x2,0x11)          from _h+0x29
_h (0x10,0xf)         from _g+0x2b
_g (0x11,0x20)        from _f+0x2c
_f (0x2,0xf)          from _h+0x29
_h (0xe,0xd)          from _g+0x2b
_g (0xf,0x1c)         from _f+0x2c
_f (0x2,0xd)          from _h+0x29
_h (0xc,0xb)          from _g+0x2b
_g (0xd,0x18)         from _f+0x2c
_f (0x2,0xb)          from _h+0x29
_h (0xa,0x9)          from _g+0x2b
_g (0xb,0x14)         from _f+0x2c
_f (0x2,0x9)          from _h+0x29
_h (0x8,0x7)          from _g+0x2b
_g (0x9,0x10)         from _f+0x2c
_f (0x2,0x7)          from _h+0x29
_h (0x6,0x5)          from _g+0x2b
_g (0x7,0xc)          from _f+0x2c
_f (0x2,0x5)          from _h+0x29
_h (0x4,0x3)          from _g+0x2b
_g (0x5,0x8)          from _f+0x2c
_f (0x2,0x3)          from _h+0x29
_h (0x2,0x1)          from _g+0x2b
_g (0x2,0x3)          from _f+0x2c
_f (0x1,0x1)          from _main+0x15
_main (0x1,0x187ef20,0x187ef28) from __start+0x39
```

Exit **adb** by typing:

```
$q
```


Using the adb Memory Maps

The **adb** program prepares a set of maps for the text and data segments in your program and uses these maps to access items that you request for display. The following sections describe how to view these maps, and how they are used to access the text and data segments.

Displaying the Memory Maps

You can display the contents of the memory maps using the **\$m** command. The command has the following form:

\$m [*segment*]

where *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory.

Displays for UNIX 286

If you have started **adb** but have not executed the program, the **\$m** command display has the following form:

Text Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - File
63.	160.	3712.	2462.	
Data Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - File
71.	160.	3712.	2462.	

Using the adb Memory Maps

If you have executed the program, the command displays the following form:

Text Segments			
Seg #	File Pos	Vir Size	Phys Size
63.	160.	3712.	2462.

Data Segments			
Seg #	File Pos	Vir Size	Phys Size
71.	160.	3712.	2462.

Displays for UNIX 386

The \$m command has the following form:

Text Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0

Data Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000	0xe90	0x460	0x1880000

Each entry gives the segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The filenames to the right of the display are the program and core filenames.

If you have executed the program, the command displays the following form:

Text Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0

Data Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000	0x1880e90	0x460	0x1880000

where virtual size is the number of bytes the segment occupies in memory. This size is sometimes different from the size of the segment in the file and will often change as you execute the program. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program files. The file position value is ignored.

Giving Segment Numbers

If you give a segment number with the command, **adb** displays information only about that segment. For example, the following command displays a map for segment 63 only:

```
$m 63
```

The display has the following form for XENIX 286:

```
Segment # = 63.  
Type = Text  
File position = 160.  
Virtual Size = 3712.  
Physical Size = 2048.
```

The display has the following form for XENIX 386:

```
Segment # = 0x3f  
Type = Text  
File position = 0x3f  
Virtual size = 0x400  
Physical Size = 0x400  
Reloc Base = 0x0
```


Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the following form:

?m *segment-number file-position size*

and:

/m *segment-number file-position size*

where:

- *segment-number* gives the number of the segment map you wish to change,
- *file-position* gives the offset in the file to the beginning of the given address,
- *size* gives the segment size in bytes,
- **?m** assigns values to a text segment entry, and
- **/m** assigns values to a data segment entry.

For example, the following command changes the file position for segment 0x3f in the text map to 0x2000:

```
?m 0x3f 0x2000
```

The following command changes the file position for segment 0x47 in the data map to 0x0:

```
/m 0x47 0x0
```

Creating New Map Entries

You can create new segment maps and add them to your memory map by using the **?M** and **/M** commands. Unlike **?m** and **/m**, these commands create a new map instead of changing an existing one. These commands have the following form:

?M *segment-number file-position size*

and:

/M *segment-number file-position size*

where:

- *segment-number* gives the number of the segment map you wish to create,
- *file-position* gives the offset in the file to the beginning of the given address, and
- *size* gives the segment size in bytes.

The **?M** command creates a text segment entry; **/M** creates a data segment entry. The segment number must be unique. You cannot create a new map entry that has the same number as an existing one.

The **?M** and **/M** commands are especially useful if you wish to access segments that are otherwise allocated to your program. For example, the following command creates a text segment entry for segment 0x47 whose size is 0x9c8 bytes:

```
?M 0x47 0x0 0x9c8
```

Validating Addresses

Whenever you use an address in a command, **adb** checks the address to make sure it is valid. To validate the address, **adb** uses the segment number, file position, and size values in each map entry. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

12

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the `? command must appear in the text segments map; segments used with the / command must appear in the data segments map. If the value does not belong to the map, adb displays a bad segment error.`

The next step is to check the offset to see if it is in range. The offset must be within the following range:

$$0 \leq \text{offset} \leq \text{segment-size}$$

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address like the following, then uses this effective address to read from the corresponding file:

$$\text{effective-file-address} = \text{offset} + \text{file-position}$$

Miscellaneous Features

The following sections explain a number of useful commands and features of **adb**.

Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored.

One typical combination is to place a **?** command after an **l** command. For example, the following command searches for and displays a string that begins with the characters *Th*:

```
?l 'Th' ; ?s
```

Creating adb Scripts

You can direct **adb** to read commands from a text file instead of the keyboard by redirecting **adb**'s standard input file at invocation. To redirect the standard input, use the standard redirection symbol **<** and supply a filename. For example, to read commands from the file *script*, type:

```
adb sample <script
```

The file you supply must contain valid **adb** commands. Such files are called script files, and can be used with any invocation of the debugger.

Reading commands from a script file is very convenient when you wish to use the same set of commands on several different object files. Scripts are typically used to display the contents of core files after a program error.

Miscellaneous Features

For example, you can use a file containing the following commands to display most of the relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

Setting Output Width

You can set the maximum width (in characters) of each line of output created by **adb** by using the **\$w** command. The command has the following form:

n\$w

where *n* is an integer number giving the width in characters of the display. You can give any width that is convenient for your given terminal or display device. The default width, when **adb** is first invoked, is 80 characters.

The command is typically used when redirecting output to a lineprinter or special terminal. For example, the following command sets the display width to 120 characters, a common maximum width for lineprinters:

```
120$w
```

Setting the Maximum Offset

The **adb** program normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data you are viewing with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data that are no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason, **adb** lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **\$s** command. The command has the following form where *n* is an integer giving the new offset:

n\$s

For example, the following command increases the maximum possible offset to 4095:

4095\$s

All instructions and data that are no more than 4095 bytes away are given symbolic addresses.

Note that you can disable all symbolic addressing by setting the maximum offset to zero. All addresses will be given numeric values instead.

Setting Default Input Format

You can set the default format for numbers used in commands with the **\$d** (decimal), **\$o** (octal), and **\$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with *0* or *0x*, and how to display numbers when no specific format is given.

The commands are useful if you wish to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you use the following combination, you can give addresses in hexadecimal without prepending each address with *0x*:

\$x

Furthermore, **adb** displays all numbers in hexadecimal except those that are specifically requested to be in some other format.

When you first start **adb**, the default format is decimal. You can change this at any time and restore it as necessary using the **\$d** command.

Using UNIX Commands

You can execute UNIX commands without leaving **adb** by using the **adb** escape command **!**. The escape command has the following form:

```
! command
```

where *command* is the UNIX command you wish to execute. The command must have any required arguments. The **adb** passes this command to the system shell which executes it. When finished, the shell returns control to **adb**.

For example, to display the date, type:

```
! date
```

The system displays the date at your terminal and restores control to **adb**.

Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the **=** command. This command directs **adb** to display the value of an expression in a given format.

You use the **=** command to convert numbers in one base to another, to double-check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the following command displays the hexadecimal number "0x2a" as the decimal number 42:

```
0x2a=d
```

however, the following command displays it as the ASCII character asterisk (*):

```
0x2a=c
```

Expressions in a command may have any combination of symbols and operators. You can also compute the value of external symbols, by typing:

```
main+5=X
```

This is helpful if you wish to check the hexadecimal value of an external symbol address.

Note that the `=` command can also be used to display literal strings at your terminal. This is especially useful in an **adb** script where you may wish to display comments about the script as it performs its commands. For example, the following command spaces three lines:

```
=3n"C Stack Backtrace"
```

The system then displays the following message:

```
C Stack Backtrace
```

12

An Example: Directory and Inode Dumps

This example illustrates how to create **adb** scripts to display the contents of a directory file and the inode map of a UNIX file system. The directory file is assumed to be named *dir*, and contains a variety of files. The UNIX file system is assumed to be associated with the device file */dev/src*, and has the necessary permissions for you to read it.

To display a directory file, you must create an appropriate script, then start **adb** with the name of the directory, redirecting its input to the script.

First, you can create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned *inumber* and a 14-character filename. You can display this information by adding the following command to the script file:

```
0,-1?ut14cn
```

This command displays one entry for each line, separating the number and filename with a tab. The display continues to the end of the file. If you place the following command at the beginning of the script, **adb** will display the strings as headings for the columns of numbers:

```
= "inumber"8t "Name"
```

Once you have the script file, type:

```
adb dir - <script
```

Miscellaneous Features

(The dash (-) is used to prevent **adb** from attempting to open a core file.) The **adb** program reads the commands from the script and displays the following:

inumber	name
652	.
82	..
5971	cap.c
5323	cap
0	pp

To display the inode table of a file system, you must create a new script, then start **adb** with the filename of the device associated with the file system (such as the hard disk drive).

The structure of an inode table entry is defined in the file */usr/include/sys/ino.h*. Each inode entry includes:

- an unsigned short containing the mode and type of the file
- a short containing the number of links to the file
- an unsigned short containing the owner's user ID
- an unsigned short containing the owner's group ID
- a long containing the size of the file in bytes
- an array of 40 bytes containing the disk block addresses (only 39 of the 40 address bytes are in use: 13 addresses of 3 bytes each)
- a long giving the time the file was last accessed
- a long giving the time the file was last modified
- a long giving the time the file was created

The inode table starts at the address "04000" of the filesystem. This is the address of the beginning of block 2 of the file system. (For a discussion of the layout of a file system, see **filesystem(F)** of the *UNIX User's Reference*.)

You can display the first entry by typing:

```
04000, -1?onororon2un40b3Y2na
```

Several Newlines are inserted within the display to make it easier to read.

To use the script on the inode table of */dev/src*, type:

```
adb /dev/src - <script
```

(Again, the dash (-) is used to prevent an unwanted core file.) Each entry in the inode table display has the following form:

0.:2048.:	040755								
	046	03	03						
	640	0							
	0121	06	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	1986 Dec 4 13:55:49			1986 Nov 20 20:46:13			1986 Nov 20 20:46:13		

Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by using the **w** and **W** commands and invoking **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

Locating Values in a File

You can locate specific values within a file by using the **I** and **L** commands. The commands have the following form:

[*address*] ?**I** *value*

[*address*] ?**L** *value*

where:

- *address* is the address at which to start the search,
- **I** command searches for 2 byte values,
- **L** command searches for 4 byte values, and
- *value* is the value (given as an expression) to be located.

The following command starts the search at the current address, and continues until the first match or the end of the file:

?**I**

If the value is found, the current address is set to that value's address.

Writing to a File

You can write to a file by using the **w** and **W** commands. The commands have the following form:

```
[ address ] ?w value
```

```
[ address ] ?W value
```

where:

- *address* is the address of the value you wish to change,
- **w** command writes 2 byte values,
- **W** writes 4 bytes, and
- *value* is the new value.

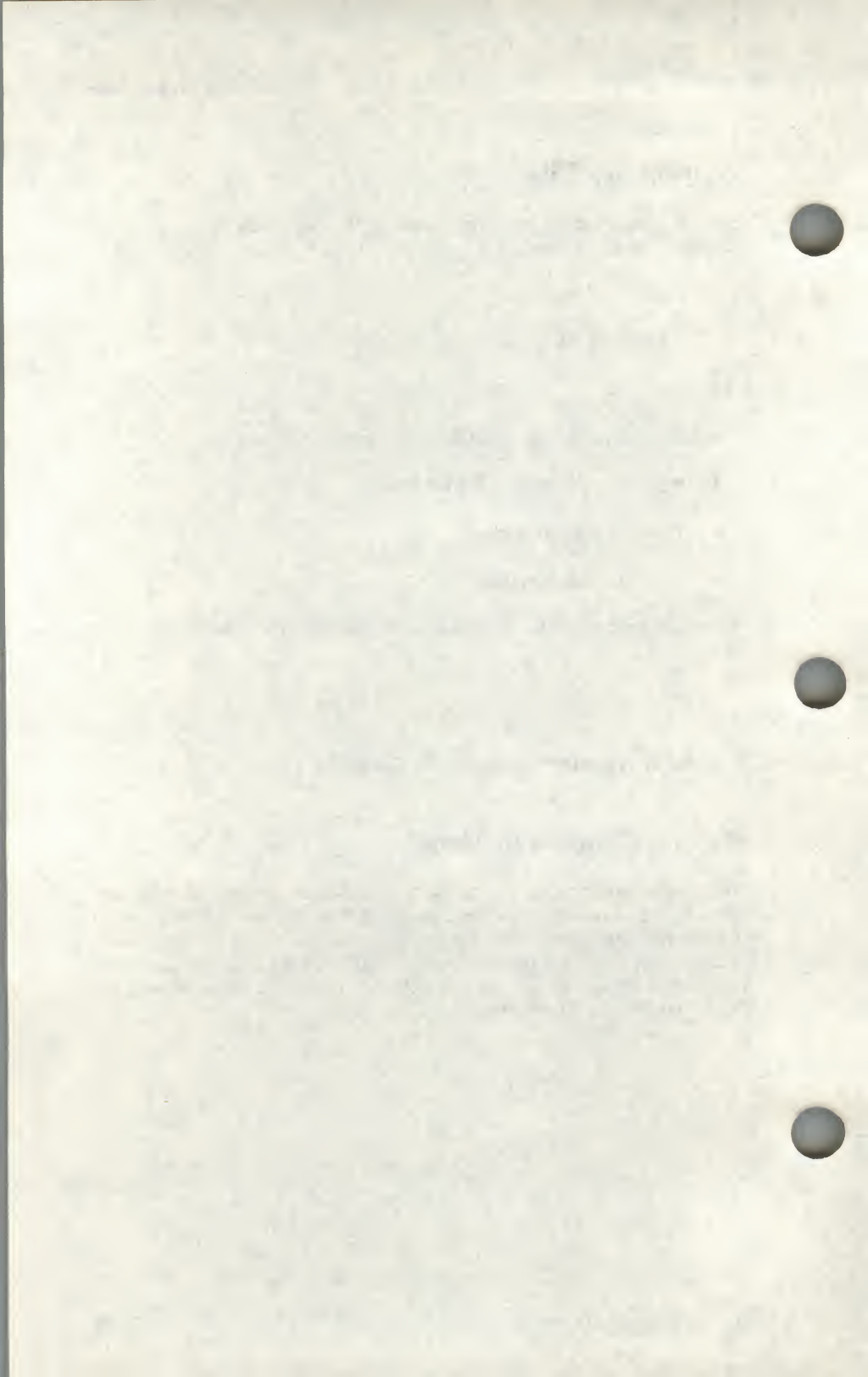
For example, the following commands change the word “This” to “The”:

```
?l 'Th'
?W 'The'
```

Note that **W** is used to change all four characters.

Making Changes to Memory

You can also make changes to memory whenever a program has been executed. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program’s data as it runs, for example, to change the value of program flags or constants temporarily.



Chapter 13

File and Record Locking

Introduction 13-1

Terminology 13-2

File Protection 13-4

 Opening a File for Record Locking 13-4

 Setting a File Lock 13-5

 Setting and Removing Record Locks 13-7

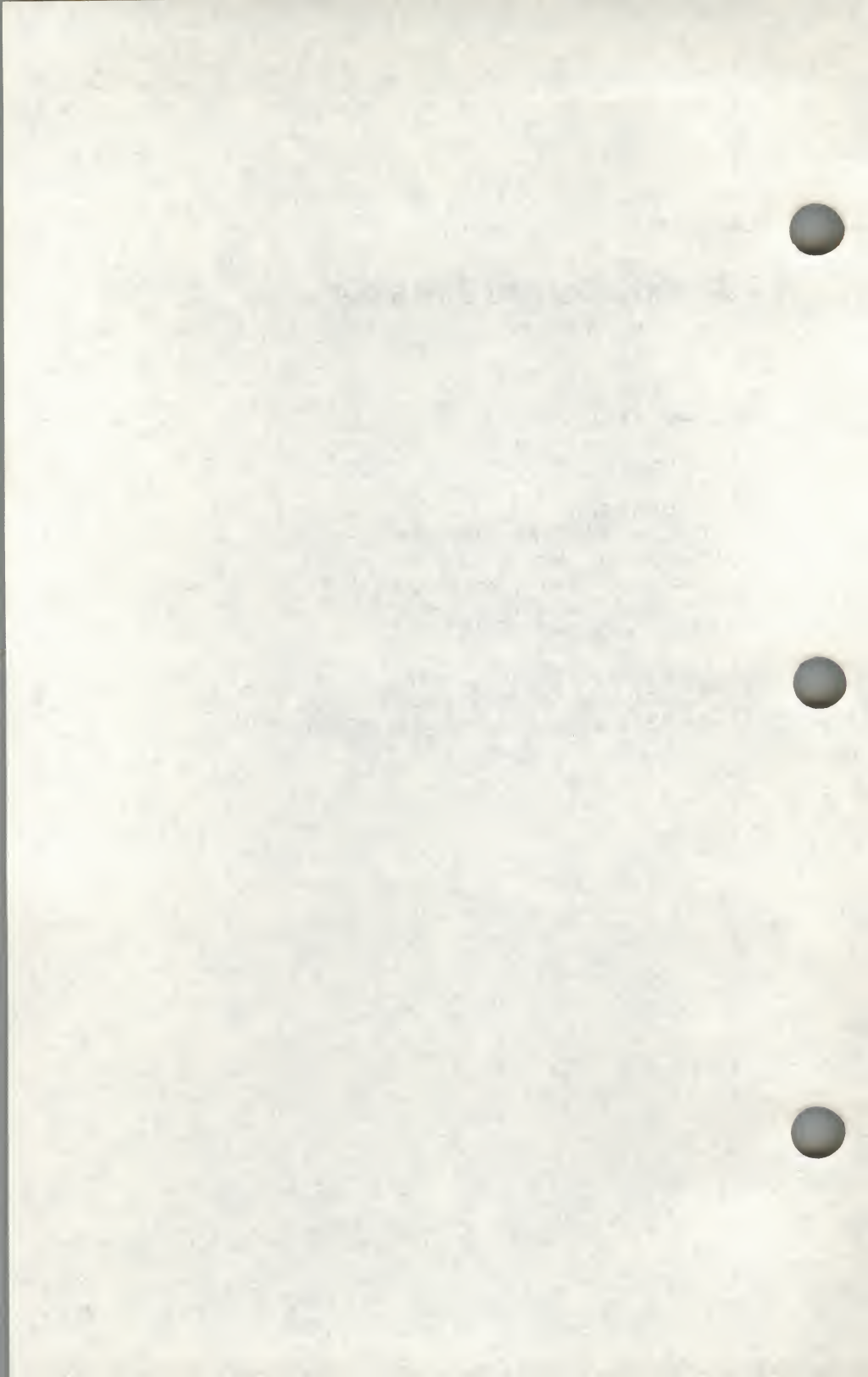
 Getting Lock Information 13-12

 Deadlock Handling 13-15

Selecting Advisory or Mandatory Locking 13-16

 Caveat Emptor—Mandatory Locking 13-17

 Record Locking and Future Releases of UNIX 13-17



Introduction

Both mandatory and advisory file and record locking are available on current releases of the UNIX System. This capability is intended to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multiuser applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX System users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking prevents processes from accessing data that has been locked by another process.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl(S)** system call, the **lockf(S)** library function, and **fcntl(M)** data structures and commands are referred to throughout this section. You should read them in the *Programmer's Reference* before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record

This is a contiguous set of bytes in a file. The UNIX Operating System does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes

These are processes that work together in some well-defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock, it may assume that no other process will be writing or updating that record at the same time. Read locks also permit many processes to read the given record. This is useful when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read lock or write lock that record, in whole or in part. If a process holds a write lock, it may assume that no other process will be reading or writing that record until the process holding the write lock unlocks the record.

Advisory Locking

This is a form of record locking that does not interact with the I/O subsystem [that is, **creat(S)**, **open(S)**, **read(S)**, and **write(S)**]. The control over records is accomplished by using an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request by the I/O subsystem.

Mandatory Locking

This is a form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat(S)**, **open(S)**, **read(S)**, and **write(S)** system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX System files to control who may read, write, or execute a file. These access permissions may only be set by the owner of the file or by the super-user. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write, or execute permission for that user. Any information that is worth protecting is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

For example, only a known set of programs and/or administrators should be able to read or write a database. This can be done easily by setting the ownership and the set-group-ID bit [see **chmod(C)** and **chown(C)** in the *User's Reference*] of the database accessing programs and correct permissions on the database data files. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail(C)** command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility, and the same is true for write locks and write accessibility. For our example we will open our file for both read and write access:

```

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

int fd;          /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
    extern void exit(), perror();

    /* get database file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
    .
    .
    .

```

13

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl(S)** system call, the other using the */usr/group* standards compatible **lockf(S)** library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) and ending at the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl(S)** system call is as follows:

File Protection

```
#include <fcntl.h>
#define MAX_TRY 10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address of
 * which is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK; /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L; /* until the end of the file address space */

/* Attempt locking MAX_TRY times before giving up.
 */
while (fcntl(fd, F_SETLK, &lck) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again, such as ENOLCK.
         */
        if (++try < MAX_TRY) {
            (void) sleep(5);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("fcntl");
    exit(1);
}
.
.
.
```

This portion of code tries to lock a file. This task is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because MAX_TRY has been exceeded.

To perform the same task using the **lockf(S)** function, the code is as follows:


```

#include <unistd.h>
#define MAX_TRY    10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
if (lseek(fd, 0L, 0) == -1);
{
    perror("lseek");
    exit(1);
}
/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
    if (errno == EAGAIN || errno == EACCES) {
        /* there might be other errors cases in which
         * you might try again.
         */
        if (++try < MAX_TRY) {
            sleep(5);
            continue;
        }
        (void) fprintf(stderr, "File busy try again later!\n");
        return;
    }
    perror("lockf");
    exit(2);
}
.
.
.

```

13

It should be noted that the **lockf(S)** example appears to be simpler, but the **fcntl(S)** example exhibits additional flexibility. Using the **fcntl(S)** method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf(S)** merely sets write (exclusive) locks, and an additional system call [**lseek(S)**] is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the inter-record pointers in a doubly linked list.) To do this you must decide the following

questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might do the following:

- wait a certain amount of time and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above.

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```

struct record {
    .
    .      /* data portion of record */
    .
    long prev;      /* index to previous record in the list */
    long next;      /* index to next record in the list */
};

/* Lock promotion using fcntl(S)
 * When this routine is entered it is assumed that there are
 * read locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *   Set a write lock on "this".
 *   Return index to "this" record.
 * If any write lock is not obtained:
 *   Restore read locks on "here" and "next".
 *   Remove all other locks.
 *   Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
    struct flock lck;

    lck.l_type = F_WRLCK; /* setting a write lock */
    lck.l_whence = 0; /* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* promote lock on "here" to write lock */
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "this" failed;
         * demote lock on "here" to read lock.
         */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Lock on "next" failed;
         * demote lock on "here" to read lock,
         */

```


File Protection

```
lck.l_type = F_RDLCK;
lck.l_start = here;
(void) fcntl(fd, F_SETLK, &lck);
/* and remove lock on "this".
 */
lck.l_type = F_UNLCK;
lck.l_start = this;
(void) fcntl(fd, F_SETLK, &lck);
return (-1); /* cannot set lock, try again or quit */
}

return (this);
}
```

13

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the `F_SETLKW` command. If the `F_SETLK` command was used instead, the `fcntl` system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the `lockf` function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```

/* Lock promotion using lockf(S)
 * When this routine is entered, it is assumed that there are
 * no locks on "here" and "next".
 * If locks are obtained:
 *   Set a lock on "next".
 *   Return index to "this" record.
 * If any lock is not obtained:
 *   Remove all other locks.
 *   Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;
{
    /* lock "here" */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* lock "this" */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
    }

    /* lock "next" */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

        /* Lock on "next" failed.
         * Clear lock on "here",
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));

        /* and remove lock on "this".
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1); /* cannot set lock, try again or quit */
    }

    return (this);
}

```

Locks are removed in the same manner as they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

13 One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous **fcntl** example and the F_GETLK command is used in the **fcntl** call to test for a lock. If the lock passed to **fcntl** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl**. That is, the lock data passed to **fcntl** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture, the value in **l_sysid** should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process' lock, then the **l_type** field is changed to F_UNLCK, and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment, only one of these will be found.


```

struct flock lck;

/*Find and print "write lock"blocked segments of this file.*/
(void) printf("sysid pid type start length\n");
lck.l_whence = 0;
lck.l_start = 0L;
lck.l_len = 0L;
do {
    lck.l_type = F_WRLCK;
    (void) fcntl(fd, F_GETLK, &lck);
    if (lck.l_type != F_UNLCK) {
        (void) printf("%5d %5d %c %8d %8d\n",
            lck.l_sysid,
            lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R',
            lck.l_start,
            lck.l_len);
        /* if this lock goes to the end of the address
         * space, no need to look further, so break out.
         */
        if (lck.l_len == 0)
            break;
        /* otherwise, look for new lock after the one
         * just found.
         */
        lck.l_start += lck.l_len;
    }
} while (lck.l_type != F_UNLCK);

```

13

The **fcntl** function with the **F_GETLK** command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf** function with the **F_TEST** command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **lockf** to test for a lock on a file follows:

File Protection

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unknown error <%d>\n", errno);
            break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using an **l_whence** value of 1. If both the parent process and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf(S)** function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This indicates that the relative offset, **l_start** bytes, will be measured from the start of the file, or the end of the file, respectively.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection, it should set its locks using **F_SETLK** instead of **F_SETLKW**.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and by the state of the permissions on the file [see **chmod(S)**]. For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group-execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;

.
.
.
if (stat(filename, &buf) < 0) {
    perror("program");
    exit (S);
}
/* get currently set mode */
mode = buf.st_mode;
/* remove group execute permission from mode */
mode &= ~(S_IXGRP);
/* set 'set group id bit' in mode */
mode |= S_ISGID;
if (chmod(filename, mode) < 0) {
    perror("program");
    exit(1);
}

.
.
.
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod(C)** command can also be easily used to set a file to have mandatory locking. This can be done with the command,

```
chmod +l filename
```

The **ls(C)** command was also changed to show this setting when you ask for the long listing format:

```
ls -l filename
```

causes the following to be printed:

```
-rw---l--- 1 abc other 1048576 Dec 3 11:44 filename
```

13

Caveat Emptor—Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX System file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of UNIX

Provisions have been made for file and record locking in a UNIX System environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system.

Selecting Advisory or Mandatory Locking

Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the **sleep-when-blocked** feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Chapter 14

Shared Libraries

Introduction 14-1

Using a Shared Library 14-2

What is a Shared Library? 14-2

The UNIX System Shared Libraries 14-3

Building an **a.out** File 14-4

Coding an Application 14-5

Deciding Whether to Use a Shared Library 14-5

More About Saving Space 14-6

How Shared Libraries Save Space 14-6

How Shared Libraries Are Implemented 14-9

How Shared Libraries Might Increase Space Usage 14-11

Identifying **a.out** Files that Use Shared Libraries 14-12

Debugging **a.out** Files that Use Shared Libraries 14-12

Building a Shared Library 14-13

The Building Process 14-13

Step 1: Choosing Region Addresses 14-13

Step 2: Choosing the Target Library Pathname 14-16

Step 3: Selecting Library Contents 14-16

Step 4: Rewriting Existing Library Code 14-16

Step 5: Writing the Library Specification File 14-16

Step 6: Using **mkshlib** to Build the Host and Target 14-19

Guidelines for Writing Shared Library Code 14-21

Choosing Library Members 14-22

Changing Existing Code for the Shared Library 14-24

Using the Specification File for Compatibility 14-26

Importing Symbols 14-28

Referencing Symbols in a Shared Library from Another

Shared Library 14-34

Providing Archive Library Compatibility 14-36

Tuning the Shared Library Code 14-37

Checking for Compatibility 14-40

Checking Versions of Shared Libraries Using

chkshlib(CP) 14-40

An Example 14-44

 The Original Source 14-44

 Choosing Region Addresses and the Target Pathname 14-48

 Selecting Library Contents 14-48

 Rewriting Existing Code 14-48

 Writing the Specification File 14-50

 Building the Shared Library 14-51

 Using the Shared Library 14-51

Summary 14-52

Introduction

Efficient use of disk storage space, memory, and computing power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make **a.out** files (executable object files) smaller on disk storage and processes (**a.out** files that are executing) smaller in memory.

The first part of this chapter, “Using a Shared Library,” is designed to help you use UNIX System V shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, “Building a Shared Library,” describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers; advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the UNIX System tool **mkshlib**(CP) (documented in the *Programmer's Reference*) and how to write C code for shared libraries on a UNIX System. An example is included. This part also describes how to use the tool **chkshlib**(CP), which helps you check the compatibility of versions of shared libraries. Read this part of the chapter only if you have to build a shared library.

Note

Shared libraries are a new feature of UNIX System V Release 3.0 and later. An executable object file that needs shared libraries will not run on previous releases of UNIX System V.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains what shared libraries are and how and when to use them on the UNIX System.

What is a Shared Library?

14 A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX System executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

The implementation behind these concepts is a shared library with two pieces. The first, called the host shared library, is an archive that the link editor searches to resolve user references and to create the **.lib** section in **a.out** files. The structure and operation of this archive is the same as any archive without shared library members. For simplicity, however, in this chapter references to archives mean archive libraries without shared library members.

The second part of a shared library is the target shared library. This is the file that the UNIX System uses when running **a.out** files built with the host shared library. It contains the actual code for the routines in the library. Naturally, it must be present on the the system where the **a.out** files will be run.

A shared library offers several benefits by not copying code into **a.out** files. It can

- save disk storage space

Because shared library code is not copied into all the **a.out** files that use the code, **a.out** files are smaller and use less disk space.

- save memory

By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

“Deciding Whether to Use a Shared Library” in this chapter describes shared libraries in more detail.

The UNIX System Shared Libraries

Shared libraries are part of the SDS core. The networking library included with the Networking Support Utilities is also a shared library. Other shared libraries may be available now from software vendors and in the future from AT&T.

Shared Library	Host Library Command Line Option	Target Library Pathname
C Library	<code>-lc_s</code>	<code>/shlib/libc_s</code>
Networking Library	<code>-lnsl_s</code>	<code>/shlib/libnsl_s</code>

Notice the `_s` suffix on the library names; we use it to identify both host and target shared libraries. For example, it distinguishes the standard relocatable C library `libc` from the shared C library `libc_s`. The `_s` also indicates that the libraries are statically linked.

The relocatable C library is still available with releases of the C Programming Language Utilities; this library is searched by default during the compilation or link editing of C programs. All other archive libraries from previous releases of the system are also available. Just as you use the archive libraries' names, you must use a shared library's name when you want to use it to build your `a.out` files. You tell the link editor its name with the `-l` option, as shown below.

Using a Shared Library

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the UNIX System:

```
cc file.c -o file ... -llibrary_file ...
```

To direct a search of the networking library, for example, you use the following command line.

```
cc file.c -o file ... -lnsl_s ...
```

And to link all the files in your current directory together with the shared C library you'd use the following command line:

```
cc *.c -lc_s
```

Normally, you should include the **-lc_s** argument after all other **-l** arguments on a command line. The shared C library will then be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line are searched.

A shared library might be built with references to other shared libraries. That is, the first shared library might contain references to symbols that are resolved in a second shared library. In this case, both libraries must be given on the **cc** command line, in the order of the dependencies.

For example, if the library **libX.a** references symbols in the shared C library, the command line would be as follows:

```
cc *.c -lX_s -lc_s
```

Notice that the shared library containing the references to symbols must be listed on the command line before the shared library needed to resolve those references. For more information on inter-library dependencies, see the section "Referencing Symbols in a Shared Library from Another Shared Library" later in this chapter.

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

Although there are exceptions, you should avoid redefining standard library routines, such as **printf(S)** and **strcmp(S)**. Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

Use only the functions and data mentioned on the manual pages describing the routines in Section (S) of the *Programmer's Reference*.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program. A well-designed shared library almost always saves space. So, as a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section, "Coding an Application.") Then compare the two versions of the application for size and performance. For example,

Using a Shared Library

```
$ cat hello.c
main()
{
    printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
unshared: 8680 + 1388 + 2248 = 12316
shared: 300 + 680 + 2248 = 3228
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

When making your decision about using shared libraries, also remember that they are not available on UNIX System V releases prior to Release 3.0. If your program must run on previous releases, you will need to use archive libraries.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the UNIX System
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside, and made visible outside, the library are external symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the **a.out** file.

Note

Note that the link editor on the UNIX System is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

Although these similarities exist, a shared library differs significantly from an archive library. The major differences are related to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the UNIX System handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor copies from the shared library into the program's object file only a small amount of code for initialization of imported symbols. (See the section "Importing Symbols" later in the chapter for more details on imported symbols.) For the bulk of the library code, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the UNIX System executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

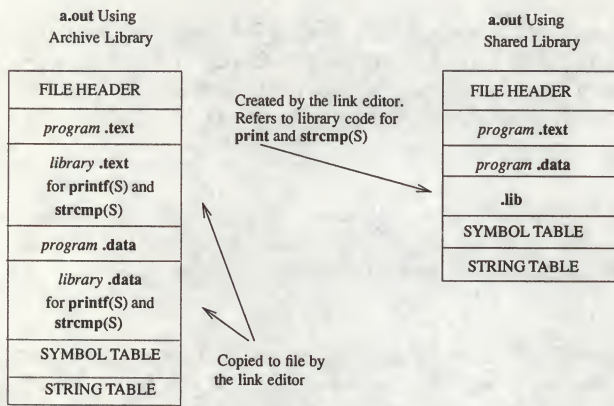
Figure 14-1 depicts the **a.out** files produced using a regular archive version and a shared version of the standard C library to compile the following program:

```
main()
{
    ...
    printf( "How do you like this manual?\n" );
    ...
    result = strcmp( "I do.", answer );
    ...
}
```

Notice that the shared version is smaller. Figure 14-2 depicts the process images in memory of these two files when they are executed.

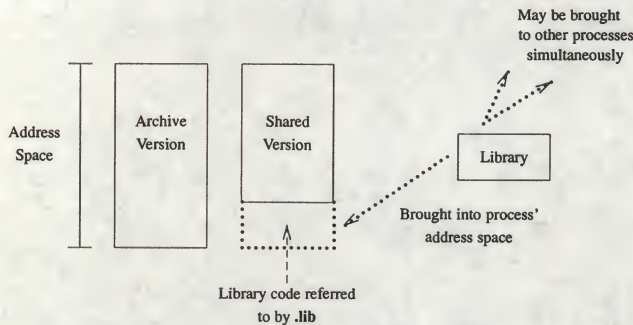
Using a Shared Library

Figure 14-1 **a.out** Files Created Using an Archive Library and a Shared Library



Now consider what happens when several **a.out** files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as indicated in Figure 14-2. This separation enables all processes using the same shared library to reference a single copy of the code.

Figure 14-2 Processes Using an Archive and a Shared Library



How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation on the UNIX System to understand how they might increase space usage (this happens seldomly). The following paragraphs describe host and target shared libraries, the branch table, and then, how shared libraries might increase space usage.

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 14-1 as **.lib**.

The target library used for execution resembles an **a.out** file. The UNIX Operating System reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the UNIX System executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the UNIX System uses the same physical code for each process that attaches a shared library's text.

Using a Shared Library

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). Processes that share text do not share data and stack area in order that they do not interfere with one another.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Processes must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

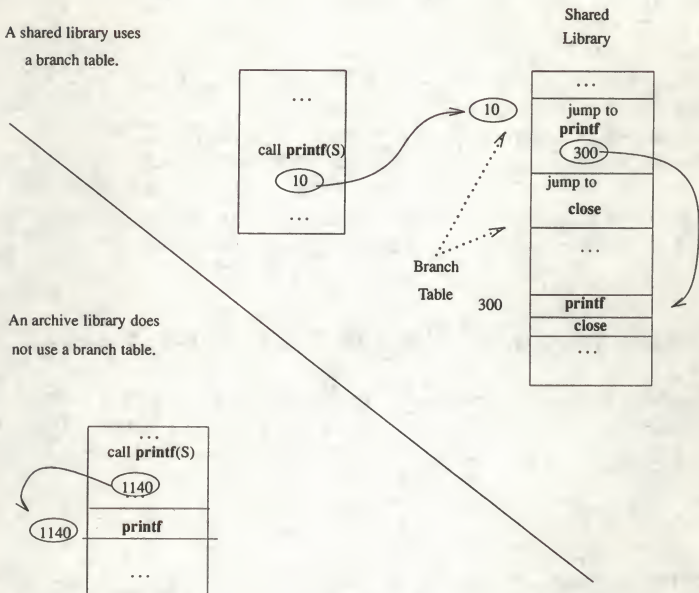
When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

14

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To prevent the need to recompile, a shared library is implemented with a branch table on the UNIX System. A branch table associates text symbols with absolute addresses that do not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 14-3 shows two **a.out** files executing a call to **printf(S)**. The process on the left was built using an archive library. It already has a copy of the library code defining the **printf(S)** symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

Figure 14-3 A Branch Table in a Shared Library

Data symbols do not have a mechanism to prevent a change of address between shared libraries. The tool **chkshlib(CP)** compares **a.out** files with a shared library to check compatibility and help you decide if the files need to be recompiled. See “Checking Versions of Shared Libraries Using **chkshlib(CP)**.”

How Shared Libraries Might Increase Space Usage

A target library might add space to a process. Recall from “How Shared Libraries are Implemented” in this chapter that a shared library’s target file may have both text and data regions connected to a process. While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process’s memory requirements. As a result, if an application uses only a small part of a shared library’s text and data, executing the application might require more memory with a shared library than without one. For example, it would be unwise to use the shared C library to access only

Using a Shared Library

strcmp(S). Although sharing **strcmp(S)** saves disk storage and memory, the memory cost for sharing all the shared C library's private data region outweighs the savings. The archive version of the library would be more appropriate.

A host library might add space to an **a.out** file. Recall that UNIX System V Release 3.0 uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. To resolve these references, the link editor has to add to the **a.out** initialization code defining the referenced imported symbols file. This code increases the size of the **a.out** file.

Identifying a.out Files that Use Shared Libraries

14

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **dump(CP)** command (documented in the *Programmer's Reference*) to look at the section headers for the file:

```
dump -hv a.out
```

If the file has a **.lib** section, a shared library is needed. If the **a.out** does not have a **.lib** section, it does not use shared libraries.

To display the libraries used by **a.out**, use the **-L** option as shown in the following example:

```
dump -L a.out
```

Debugging a.out Files that Use Shared Libraries

sdb reads the shared libraries' symbol tables and performs as documented (in the *Programmer's Reference*) using the available debugging information. The branch table is hidden so that functions in shared libraries can be referenced by their names, and the **M** command lists the names of shared libraries' target files used by the executable file, among other information.

Shared library data are not dumped to core files, however. If you encounter an error that results in a core dump and does not appear to be in your application code, you may find debugging easier if you rebuild the application with the archive version of the library used.

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps in the building process, the use of the UNIX System tool **mkshlib**(CP) that builds the host and target libraries, and some guidelines for writing shared library code. There is an example at the end of this part which demonstrates the major features of **mkshlib** and the steps in the building process.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use the UNIX System shared libraries or other shared libraries that have already been built.

The Building Process

To build a shared library on the UNIX System, you have to complete six major tasks:

- choosing region addresses
- choosing the pathname for the shared library target file
- selecting the library contents
- rewriting existing library code to be included in the shared library
- writing the library specification file
- using the **mkshlib** tool to build the host and target libraries

Here each of these tasks is discussed.

Step 1: Choosing Region Addresses

The first thing you need to do is choose region addresses for your shared library.

Shared library regions on the 386-based computer correspond to memory management unit (MMU) segment size, each of which is 4 MB. The following table gives a list of the segment assignments on the 386-based computer (as of the copyright date for this guide) and shows what virtual addresses are available for libraries you might build.

Building a Shared Library

Start Address	Contents	Target Pathname
0xA0000000 . . . 0xA3C00000	Reserved for AT&T UNIX System Shared C Library AT&T Networking Library	 /shlib/libc_s /shlib/libnsl_s
0xA4000000 0xA4400000 0xA4800000 0xA4C00000	Generic Database Library	Unassigned
0xA5000000 0xA5400000 0xA5800000 0xA5C00000	Generic Statistical Library	Unassigned
0xA6000000 0xA6400000 0xA6800000 0xA6C00000	Generic User Interface Library	Unassigned
0xA7000000 0xA7400000 0xA7800000 0xA7C00000	Generic Screen Handling Library	Unassigned
0xA8000000 0xA8400000 0xA8800000 0xA8C00000	Generic Graphics Library	Unassigned
0xA9000000 0xA9400000 0xA9800000 0xA9C00000	Generic Networking Library	Unassigned
0xAA000000 . . . 0xAFC00000	Generic - to be defined	Unassigned
0xB0000000 . . . 0xBF000000	For private use	Unassigned

What does this table tell you? First, the UNIX System shared C library and the networking library reside at the pathnames given above and use addresses in the reserved range. If you build a shared library that uses reserved addresses you run the risk of conflicting with future products.

Second, a number of segments are allocated for shared libraries that provide various services such as graphics, database access, and so on. These categories are intended to reduce the chance of address conflicts among commercially available libraries. Although two libraries of the same type may conflict, that doesn't matter. A single process should not usually need to use two shared libraries of the same type. If the need arises, a program can use one shared library and one archive library.

Note

Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

Third, several segments are reserved for private use. If you are building a large system with many **a.out** files and processes, shared libraries might improve its performance. As long as you don't intend to release the shared libraries as separate products, you should use the private region addresses. You can put your shared libraries into these segments and avoid conflicting with commercial shared libraries. You should also use these segments when you will own all the **a.out** files that access your shared library. Don't risk address conflicts.

Note

If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the UNIX System without shared library support.

Building a Shared Library

Step 2: Choosing the Target Library Pathname

After you choose the region addresses for your shared library, you should choose the pathname for the target library. We chose `/shlib/libc_s` for the shared C library and `/shlib/libnsl_s` for the networking library. (As mentioned earlier, we use the `_s` suffix in the pathnames of all statically linked shared libraries.) To choose a pathname for your shared library, consult the established list of names for your computer or see your system administrator. Also keep in mind that shared libraries needed to boot a UNIX System should normally be located in `/shlib`; other application libraries normally reside in `/usr/lib` or in private application directories. Of course, if your shared library is for personal use, you can choose any convenient pathname for the target library.

Step 3: Selecting Library Contents

14 Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections: "Importing Symbols," "Referencing Symbols in a Shared Library from Another Shared Library," and "Tuning the Shared Library Code."

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the section "Changing Existing Code for the Shared Library" in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is given in the section towards the end of the chapter, "An Example." Also, see the section "Using the Specification File for Compatibility" in this chapter. The contents and format of the specification file are given by the following directives (see also the **mkshlib**(CP) manual page).

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

#address *sectname address*

Specifies the start address, *address*, of section *sectname* for the target file. This directive is typically used to specify the start addresses of the *.text* and *.data* sections.

#target *pathname*

Specifies the pathname, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute pathname, but it does not have to be.

This directive must be specified exactly once per specification file.

#branch

Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

funcname is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one. The same position cannot be specified more than once, and every position from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as empty slots and can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive must be specified exactly once per shared library specification file.

#objects

Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by a newline character. This list of objects will be used to build the shared library.

This directive must be specified exactly once per shared library specification file.

#objects noload

Specifies the ordered list of host shared libraries to be searched to resolve references to symbols not defined in the library being built and not imported. Resolution of a reference in this way requires a version of the symbol with an absolute address to be found in one of the listed libraries. It is considered an error if a non-shared version of a symbol is found during the search for a shared version of the symbol.

Each name specified is assumed to be a pathname to a host or an argument of the form **-lX**, where **libX.a** is the name of a file in the default library locations. This behavior is identical to that of **ld**, and the **-L** option can be used on the command line to specify other directories in which to locate these archives.

#init object

Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

```
ptr <white space> import
```

ptr is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each such line is of the form:

```
ptr = &import;
```

All initializations for a particular object file must be given once, and multiple specifications of the same object file are not allowed.

#hide linker [*]

This directive changes symbols that are normally external into static symbols, local to the library being created. A regular expression may be given [**sh**(C), **egrep**(C) in the *User's Reference*], in which case all external symbols matching the regular expression are hidden; the **#export** directive can be used to counter this effect for specified symbols.

The optional "*" is equivalent to the directive

```
#hide linker
      *
```

and causes all external symbols to be made into static symbols.

All symbols specified in **#init** and **#branch** directives are assumed to be external symbols, and cannot be changed into static symbols using the **#hide** directive.

#export linker [*]

Specifies those symbols that a regular expression in a **#hide** directive would normally cause to be hidden but that should nevertheless remain external. For example,

```
#hide linker *
#export linker
      one
      two
```

causes all symbols except one, two, and those used in **#branch** and **#init** entries to be tagged as static.

#ident string

Specifies a string, *string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library.

##

Specifies a comment. The rest of the line is ignored.

Step 6: Using mkshlib to Build the Host and Target

The UNIX System command **mkshlib**(CP) builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(CP), and link editor, **ld**(CP). Tools are invoked through the use of **execvp** [see **exec**(S)], which searches directories in a user's **\$PATH** environment variable. Also, prefixes to **mkshlib** are parsed in much the same manner as prefixes to the **cc**(CP) command and invoked tools are given the prefix,

Building a Shared Library

where appropriate. For example, **3bmkshlib** invokes **3bld**. These commands all are documented in the *Programmer's Reference*.

The user input to **mkshlib** consists of the library specification file and command line options. The shared library build tool has the following syntax:

```
mkshlib -s specfil -t target [-h host] [-L dir...] [-n] [-q]
```

- s specfil** Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library.
- t target** Specifies the name, *target*, of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section “Writing the Library Specification File”). If the **-n** option is given, then a new target shared library will not be generated.
- h host** Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.
- n** Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **-t** option must still be supplied since a version of the target shared library is needed to build the host shared library.
- L dir** Changes the algorithm of searching for the host shared libraries specified with the **#objects noload** directive to cause the directories in *dir* to be searched before the default directories. The **-L** option can be specified multiple times on the command line, in which case the directories given with the **-L** options are searched in the order given on the command line, before the default directories.
- q** Suppresses the printing of warning messages.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility. When appropriate, we describe our experience with building the shared C library to illustrate the ways that following a particular guideline helped us.

We recommend that you read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library, then use it as a checklist to guide your planning and decision-making.

Before we consider these guidelines, let's consider the restrictions to building a shared library common to all the guidelines. These restrictions involve static linking. Here's a summary of them, some of which are discussed in more detail later. Keep them in mind when reading the guidelines in this section.

- External symbols have fixed addresses.

If an external symbol moves, you have to re-link all **a.out** files that use the library. This restriction applies both to text and data symbols.

Use of the **#hide** directive to limit externally visible symbols can help avoid problems in this area. (See "Use **#hide** and **#export** to Limit Externally Visible Symbols" in the "Using the Specification File for Compatibility" section for more details).

- If the library's text changes for one process at run time, it changes for all processes.
- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.
- Imported symbols cannot be referenced directly.

Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

Building a Shared Library

Choosing Library Members

Include Large, Frequently Used Routines

Large, frequently used routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code. **printf(S)** and related C library routines (which are documented in the *Programmer's Reference*) are good examples.

Note

Since the **printf(S)** family of routines is used frequently, we included **printf(S)** and related routines when we built the shared C library.

14

Exclude Infrequently Used Routines

Putting infrequently used routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality" in the "Tuning the Shared Library Code" section.

Exclude Routines that Use Much Static Data

Routines that use much static data increase the size of processes. As “How Shared Libraries are Implemented” and “Deciding Whether to Use a Shared Library” have explained, every process that uses a shared library gets its own private copy of the library’s data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

For example, `getgrent(S)`, which is documented in the *Programmer’s Reference*, is not used by many standard UNIX System commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library. You can import global data, if necessary, but not local, static data.

Exclude Routines that Complicate Maintenance

All external symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don’t have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of external data may affect symbol addresses and break compatibility.

Include Routines the Library Itself Needs

It usually pays to make the library self-contained. For example, `printf(S)` requires much of the standard I/O library. A shared library containing `printf(S)` should contain the rest of the standard I/O routines, too.

Note

This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Building a Shared Library

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data.

Minimize Global Data

All external data symbols are, of course, visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

14

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant. See "Use **#hide** and **#export** to Limit Externally Visible Symbols" in the section "Using the Specification File for Compatibility" later in this chapter for further tips.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new external variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose external variables were scattered throughout the library modules. Then external and static data would be intermixed. Changing static data, such as a string, like **hello** in the following example, moves subsequent data symbols, even the external symbols:

Before	Broken Successor
...	...
int head = 0;	int head = 0;
...	...
func()	func()
{	{
...	...
p = "hello";	p = "hello, world";
...	...
}	}
...	...
int tail = 0;	int tail = 0;
...	...

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus might be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

Note

The compilation system sometimes defines and uses static data invisibly to the user (e.g. tables for switch statements). Therefore, it is a mistake to assume that because you declare no static data in your shared library that you can ignore the guideline in this section.

Adding new external variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all external data symbols and place them at lower addresses than the static (hidden) data.

Building a Shared Library

You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
data1.o
...
lastdata.o
text1.o
text2.o
...
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all external variables' definitions in a single source file without a space penalty.

14

Initialize Global Data

Initialize external variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. **mkshlib** will give a fatal error if it finds an uninitialized external symbol.

Using the Specification File for Compatibility

The way in which you use the directives in the specification file can affect compatibility across versions of a shared library. This section gives some guidelines on how to use the directives **#branch**, **#hide**, and **#export**.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to re-link all of the **a.out** files that used a previous version of the library.

Use **#hide** and **#export** to Limit Externally Visible Symbols

Sometimes variables (or functions) must be referenced from several object files to be included in the shared library and yet are not intended to be available to users of the shared library. That is, they must be external so that the link editor can properly resolve all references to symbols and create the target shared library, but should be hidden from the user's view to prevent their use. Such unintended and unwanted use can result in compatibility problems if the symbols move or are removed between versions of the shared library.

The **#hide** and **#export** directives are the key to resolving this dilemma. The **#hide** directive causes **mkshlib**, after resolving all references within the shared library, to alter the symbol tables of the shared library so that all specified external symbols are made static and inaccessible from user code. You can specify the symbols to be so treated either individually and/or through the use of regular expressions.

The **#export** directive allows you to specify those symbols in the range of an accompanying **#hide** directive regular expression which should remain external. It is simply a convenience.

14

Note

It is a fatal error to try to explicitly name the same symbol in a **#hide** and an **#export** directive. For example, the following would result in a fatal error.

```
#hide linker
      one
#export linker
      one
```

#export may seem like an unnecessary feature since you could avoid specifying in the **#hide** directive those symbols that you do not want to be made static. However, its usefulness becomes apparent when the shared library to be built is complicated, and there are many symbols to be made static. In these cases, it is more efficient to use regular expressions to make all external variables static and individually list those symbols you need to be external. The simple example in the section "Writing the Library Specification File" demonstrates this point.

Note

Symbols mentioned in the **#branch** and **#init** directives are services of the shared library, must be external symbols, and cannot be made static through the use of these directives.

Note

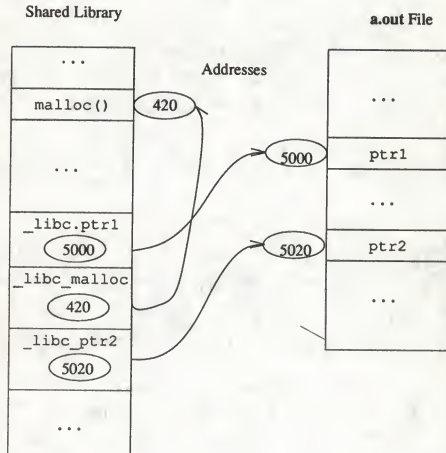
When we built the shared C library, our approach was to hide all data symbols by default, and then explicitly export symbols that we knew were needed. The advantage of this approach is that future changes to the libraries won't introduce new external symbols (possibly causing name collisions), unless we explicitly export the new symbols.

We chose the symbols to export by looking at a list of all the current external symbols in the shared C library and finding out what each symbol was used for. The symbols that were global but were only used in the shared C library were not exported; these symbols will be hidden from applications code. All other symbols were explicitly exported.

Importing Symbols

Normally, shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 14-4, the symbols **_libc.ptr1** and **_libc.ptr2** are imported from user's code and the symbol **_libc_malloc** from the library itself.

Figure 14-4 Imported Symbols in a Shared Library



The following guidelines describe when and how to use imported symbols.

Imported Symbols that the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved references to symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. Remember though that if you exclude a symbol from the target shared library that is referenced from the target shared library, you will have to import the excluded symbol.

Imported Symbols that Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most UNIX System commands use the **malloc** from the C library, they can choose either library or define their own.

Note

Three possible strategies existed for the shared C library. First, we could have excluded the **malloc(S)** family. Other library members would have needed it, and so it would have been an imported symbol. This would have worked, but it would have meant less savings.

Second, we could have included the **malloc** family and not imported it. This would have given us more savings for typical commands, but it had a price. Other library routines call **malloc** directly, and those calls could not have been overridden. If an application tried to redefine **malloc**, the library calls would not have used the alternate version. Furthermore, the link editor would have found multiple definitions of **malloc** while building the application. To resolve this the library developer would have to change source code to remove the custom **malloc**, or the developer would have to refrain from using the shared library.

Finally, we could have included **malloc** in the shared library, treating it as an imported symbol. This is what we did. Even though **malloc** is in the library, nothing else there refers to it directly; all references are through an imported symbol pointer. If the application does not redefine **malloc**, both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Let's assume a shared library wants to import the symbol `malloc`. The original archive code and the shared library code appear below.

Archive Code

```
extern char *malloc();
export ()
{
    ...
    p = malloc(n);
    ...
}
```

Shared Library Code

```
/* See pointers.c on next page */
extern char *(*_libc_malloc)();
export ()
{
    ...
    p = (*_libc_malloc)(n);
    ...
}
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The `-I` flag to `cc(CP)`, documented in the *Programmer's Reference*, would direct the C preprocessor to look in the appropriate directory to find the desired file.

Archive `import.h`

```
/* empty */
```

Shared `import.h`

```
/*
 * Macros for importing
 * symbols. One #define
 * per symbol.
 */

...
#define malloc    (*_libc_malloc)
...
extern char *malloc();
...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirect references for imported symbols. The declaration of `malloc` in `import.h` actually declares the pointer `_libc_malloc`.

Building a Shared Library

Common Source

```
#include "import.h"

extern char *malloc();

export ()
{
    ...
    p = malloc(n);
    ...
}
```

Alternatively, one can hide the `#include` with `#ifdef`:

Common Source

```
#ifdef SHLIB
#    include "import.h"
#endif

extern char *malloc();

export ()
{
    ...
    p = malloc(n);
    ...
}
```

Of course the transformation is not complete. You must define the pointer **libc_malloc**.

File **pointers.c**

```
char *(*_libc_malloc)() = 0;
```

Note that **libc_malloc** is initialized to zero, because it is an external data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of **malloc** must be assigned to **libc_malloc**. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the **a.out** file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main**. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

Note

Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into **a.out** files to set imported pointers to their correct values.

14

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libc_malloc** to the address of **malloc**:

```
#init pmalloc.o
_libc_malloc      malloc
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc;
```

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the **a.out** file. As the link editor builds the final **a.out** file, it resolves the unresolved references and collects all initialization fragments. When the **a.out** file is executed, the run time startup routines execute the initialization fragments to set the library pointers.

Building a Shared Library

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects and initialization code from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will require definitions for all three symbols, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the reference to the symbol.

Let's look at an example. In the coarse method, a single source file might define all pointers to imported symbols:

Old pointers.c

```
int (*_libc_ptr1)() = 0;
char *(*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
...
```

Allowing the loader to resolve only those references that are needed requires multiple source files and archive members. Each of the new files defines a single pointer:

File	Contents
ptr1.c	<code>int (*_libc_ptr1)() = 0;</code>
pmalloc.c	<code>char *(*_libc_malloc)() = 0;</code>
ptr2.c	<code>int (*_libc_ptr2)() = 0;</code>

Using the three files ensures that the link editor will only look for definitions for imported symbols and load in the corresponding initialization code in cases where the symbols are actually used.

Referencing Symbols in a Shared Library from Another Shared Library

At the beginning of the section "Importing Symbols," there was a statement that "normally, shared libraries cannot directly use symbols defined outside the shared library." This is true in general, and you should import all symbols defined outside the shared library whenever possible.

Unfortunately, this is not always possible, as for example when floating-point operations are performed in a shared library to be built. When such operations are encountered in any C code, the standard C compiler generates calls to functions to perform the actual operations. These functions are defined in the C library and are normally resolved in a manner invisible to the user when an **a.out** is created, since the **cc** command automatically causes the relocatable version of the C library to be searched. These floating-point routine references must be resolved at the time the shared library is being built. But, the symbols cannot be imported, because their names and usage are invisible.

The **#objects noload** directive has been provided to allow symbol references such as these to be resolved at the time the shared library is built, provided that the symbols are defined in another shared library. If there are unresolved references to symbols after the object files listed with the **#objects** directive have been link edited, the host shared libraries specified with the **#objects noload** directive are searched for absolute definitions of the symbols. The normal use of the directive would be to search the shared version of the C library to resolve references to floating-point routines.

For this use, the syntax in the specification file would be

```
#objects noload
-lc_s
```

This would cause **mkshlib** to search for the host shared library **libc_s.a** in the default library locations and to use it to resolve references to any symbols left unresolved in the shared library being built. The **-L** option can be used to cause **mkshlib** to look for the specified library in other than the default locations.

A few notes on usage are in order. When building a shared library using **#objects noload**, you must make sure that for each symbol with an unresolved reference there is a version of the symbol with an absolute definition in the searched host shared libraries, before any relocatable version of that symbol. **mkshlib** will give a fatal error if this is not the case, because relocatable definitions do not have absolute addresses and therefore do not allow complete resolution of the target shared library.

When using a shared library built with references to symbols resolved from another shared library, both libraries must be specified on the **cc** command line. The dependent library must be specified on the command line before the libraries on which it depends. (See the section "Building an **a.out** File" for more details.) If you provide a shared library which references symbols in another shared library, you should make sure that your documentation clearly states that users must specify both libraries when building **a.out** files.

Building a Shared Library

Finally, as some of the text above hints, it is possible to use **#objects noload** to resolve references to any symbols not defined in a shared library, as long as they are defined in some other shared library. We strongly encourage you to import as many symbols as possible and to use **#objects noload** only when absolutely necessary. Probably you will only need to use this feature to resolve references to floating-point routines generated by the C compiler.

Importing symbols has several important benefits over resolving references through **#objects noload**. First, importing symbols is more flexible in that it allows you to define your own version of library routines. You can define your own versions with archive versions of a library. Preserving this ability with the shared versions helps maintain compatibility.

Importing symbols also helps prevent unexpected name space collisions. The link editor will complain about multiple definitions of a symbol, references to which are resolved through the **#objects noload** mechanism, if a user of the shared library also has an external definition of the symbol.

Finally, **#objects noload** has the drawback that both the library you build and all the libraries on which it depends must be available on all the systems. Anyone who wishes to create **a.out** files using your shared library will need to use the host shared libraries. Also, the targets of all the libraries must be available on all systems on which the **a.out** files are to be run.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. Perhaps the best way to explain this guideline is by example:

Note

When we built the shared C library, We had an existing archive library to use as the base. This obviously gave us code for individual routines, and the archive library also gave us a model to use for the shared library itself.

We wanted the host library archive file to be compatible with the relocatable archive C library. However, we did not want the shared library target file to include all routines from the archive, because including them all would have hurt performance.

Reaching these goals was, perhaps, easier than you might think. We did it by building the host library in two steps. First, we used the available shared library tools to create the host library to match exactly the target. The resulting archive file was not compatible with the archive C library at this point. Second, we added to the host library the set of relocatable objects residing in the archive C library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as UNIX System V Release 3.0. The suggestions come from the experience of building the shared C library.

The archive C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a database library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Building a Shared Library

Profile the Code

To begin, profile the code that might go into the shared library (see the **prof**(CP) command in the *Programmer's Reference*).

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See “Choosing Library Members” in this chapter for more information.

Organize to Improve Locality

When a function is in an **a.out** file(s), it probably resides in a page with other code that is used more often (see “Exclude Infrequently Used Routines” in the section “Choosing Library Members”). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow**(CP) (documented in the *Programmer's Reference*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline:

Note

When we built the shared C library Using name lists and disassemblies of the shared library target file, we determined where the page boundaries fell.

After grouping related functions, we broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then we used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

After determining the branch table, we rearranged the library's object files without breaking compatibility. We put frequently used, unrelated functions together because we figured they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on. The following example shows how to change the order of the library's object files: T}

Before	After
#objects	#objects
...	...
printf.o	strcmp.o
fopen.o	malloc.o
malloc.o	printf.o
strcmp.o	fopen.o
...	...

Avoid Hardware Thrashing

You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry, again.

Building a Shared Library

Checking for Compatibility

The following guidelines explain how to check for upwardly compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, the **a.out** files will run successfully, even though versions of the library are not compatible. This may complicate development, but it is possible.

Checking Versions of Shared Libraries Using **chkshlib(CP)**

Shared library developers normally want newer versions of a library to be compatible with previous ones. As mentioned before, **a.out** files will not execute properly otherwise.

14 If you use shared libraries, you might need to find out if different versions of a shared library are compatible, or if executable files could have been built with a particular host shared library or can run with a particular target shared library. For example, you might have a new version of a target shared library, and you need to know if all the executable files that ran with the older version will run with the new one. You might need to find out if a particular target shared library can reference symbols in another shared library. A command, **chkshlib(CP)** (documented in the *Programmer's Reference*), has been provided to allow you to do these and other comparisons.

chkshlib takes names of target shared libraries, host shared libraries, and executable files as input, and checks to see if those files satisfy the compatibility criteria. **chkshlib** checks to see if every library symbol in the first file that needs to be matched exists in the second file and has the same address. The following table shows what types of files and how many of them **chkshlib** accepts as input.

The rows listed down represent the first input given, and the columns listed across represent any more inputs given. For example, if the first input file you give **chkshlib** is a target shared library, you must give another input file that is a target or host shared library.

	Nothing	Executable	Target	Host
Executable	OK	No	OK*	OK*
Target	No	No	OK	OK
Host	OK	No	OK	OK

* The executable file must be one that was built using a host shared library.

A useful way to confirm this is to use **dump -L** to find out which target file(s) gets loaded when the program is run. See **dump(CP)**, documented in the *Programmer's Reference*.

* You can also have *executable target1...targetn* and *executable host1...hostn*.

An example of a **chkshlib** command line is shown below:

```
chkshlib /shlib/libc_s /lib/libc_s.a
```

In this example, **/shlib/libc_s** is a target shared library and **/lib/libc_s.a** is a host shared library. **chkshlib** will check to see if executable files built with **/shlib/libc_s** would be able to run with **/lib/libc_s.a**.

Depending on the input it receives, **chkshlib** checks to find out if the following is true:

- an executable file will run with the given target shared library
- an executable file could have been built using the given host shared library
- an executable file produced with a given host shared library will run with a given target shared library
- an executable file that ran with an old version of a target shared library will run with a new version
- a new host shared library can replace the old host shared library; that is, executable files built with the new host shared library will run with the old target shared library
- a target shared library can reference symbols in another target shared library

Building a Shared Library

To determine if files are compatible, you have to determine which library symbols in the first file need to be matched in the second file.

- For target shared libraries, the symbols of concern are all external, defined symbols with non-zero values, except for branch labels (branch labels always start with **.bt**), and the special symbols **etext**, **edata**, and **end**.
- For host shared libraries, the symbols of concern are all external, absolute symbols with a non-zero value.
- For executable files, the symbols of concern are all external, absolute symbols with a non-zero value, except for the special symbols **etext**, **edata**, and **end**.

For two files to be compatible, the target pathnames must be identical in both files (unless the **-i** option has been specified).

14

The following table displays the output you will receive when you use **chkshlib** to check different combinations of files for compatibility. In this table **file1** represents the name of the first file given, and **file2,3,...** represents the names of any more files given as input.

Input	Output
file1 is executable file2,3,... (if any) are targets	file1 can [may not] execute using file2 file1 can [may not] execute using file3
file1 is executable file2,3 are hosts	file1 may [may not] have been produced using file2 file1 may [may not] have been produced using file3
file1 is host file2 (if any) is target	file1 can [may not] produce executables which will run with file2
file1 is target file2 is host	file2 can [may not] produce executables which will run with file1
both files are targets <i>or</i> both files are hosts	file1 can [may not] replace file2 file2 can [may not] replace file1
both files are targets and -n option is specified*	file1 can [may not] include file2

- * The **-n** option tells **chkshlib** that the two files are target shared libraries, the first of which can reference (include) symbols in the other. See “Referencing Symbols in a Shared Library from Another Shared Library” for details.

For more information on **chkshlib**, see **chkshlib(CP)**, documented in the *Programmer's Reference*.

Note

When we built the second version of the shared C library and checked it against the first version, **chkshlib** reported that many external symbols had different values and, therefore, the second version could not replace the first. Here is a list of these symbols:

<code>_bigpow</code>	<code>_qnan1.single</code>	
<code>_litpow</code>	<code>_qnan2.double</code>	
<code>_inf1.double</code>	<code>_qnan2.single</code>	
<code>_inf1.single</code>	<code>_round.double</code>	
<code>_inf2.double</code>	<code>_round.single</code>	
<code>_inf2.single</code>	<code>_trap.single</code>	
<code>_invalid.double</code>	<code>_type.double</code>	
<code>_invalid.single</code>	<code>_type.single</code>	<code>_qnan1.double</code>

Since these text symbols were not intended to be user entry points, they were not put in the branch table. So when new code was added to the shared library the addresses of these text symbols changed, and hence their values changed.

We devised the **#hide** and **#export** directives to allow us to explicitly hide the symbols we did not want to be user entry points. In fact, in the latest C Shared Library we hid all the symbols, and exported just the ones we want to be user entry points.

You cannot directly reference these functions, and these symbols will not be considered incompatible by **chkshlib** in checking the latest version of the shared C library with any subsequent version.

Dealing with Incompatible Libraries

When you determine that a newer version of a library can't replace the older version, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target pathname to the new version of the library. The host and target pathnames are independent; you don't have to change the host library pathname. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

Building a Shared Library

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library pathname for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

Note

You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

14

An Example

This section contains the process by which a small specialized shared library is created and built. We refer to the guidelines given earlier in this chapter.

The Original Source

The name of the library to be built is **libmaux** (for math auxiliary library). The interface consists of three functions, an external variable, and a header file.

The three functions:

logd	floating-point logarithm to a given base; defined in the file log.c
polyd	evaluate a polynomial; defined in the file poly.c
maux_stat	return usage counts for the other two routines in a structure; defined in stats.c ,

The external variable:

mauxerr set to non-zero if there is an error in the processing of any of the functions in the library and set to zero if there is no error (unlike **errno** in the C library),

And the header file:

maux.h declares the return types of the function and the structure returned by **maux_stat**.

The source files before any modifications for inclusion in a shared library are given below.

Figure 14-5 File log.c

```
/* log.c */
#include "maux.h"
#include <math.h>

/*
 * Return the log of "x" relative to the base "a".
 *
 *      logd(base, x) := log(x) / log(base);
 * where "log" is "log to the base E".
 */

double logd(base, x)
double base, x;
{
    extern int stats_logd;
    extern int total_calls;

    double logbase;
    double logx;

    total_calls++;
    stats_logd++;

    logbase = log((double)base);
    logx = log((double)x);
    if(logbase == -HUGE || logx == -HUGE) {
        mauxerr = 1;
        return(0);
    }
    else
        mauxerr = 0;
    return(logx/logbase);
}
```

Figure 14-6 File poly.c

```

/* poly.c */
#include "maux.h"
#include <math.h>

/* Evaluate the polynomial
 *    $f(x) := a[0] * (x^n) + a[1] * (x^{n-1}) + \dots + a[n];$ 
 * Note that there are N+1 coefficients!
 * This uses Horner's Method, which is:
 *    $f(x) := (((a[0]*x) + a[1])*x) + a[2]) + \dots + a[n];$ 
 * It's equivalent, but uses many fewer operations
 * and is more precise. */
double polyd(a, n, x)
double a[];
int n;
double x;
{
    extern int stats_polyd;
    extern int total_calls;
    double result;
    int i;

    total_calls++;
    stats_polyd++;
    if (n < 0) {
        mauxerr = 1;
        return(0);
    }
    result = a[0];
    for (i = 1; i <= n; i++)
    {
        result *= (double)x;
        result += (double)a[i];
    }
    mauxerr = 0;
    return(result);
}

```


Figure 14-7 File stats.c

```

/* stats.c */
#include "maux.h"

int total_calls = 0;
int stats_logd = 0;
int stats_polyd = 0;

int mauxerr;

/* Return structure with usage stats for functions in library
 * or 0 if space cannot be allocated for the structure */
struct mstats *
maux_stat()
{
    extern char * malloc();
    struct mstats * st;

    if((st = (struct mstats *)
        malloc(sizeof(struct mstats))) == 0)
        return(0);
    st->st_polyd = stats_polyd;
    st->st_logd = stats_logd;
    st->st_total = total_calls;
    return(st);
}

```

Figure 14-8 Header File maux.h

```

/* maux.h */

struct mstats {
    int st_polyd;
    int st_logd;
    int st_total;
};

extern double polyd();
extern double logd();
extern struct mstats * maux_stat();

extern int mauxerr;

```

Building a Shared Library

Choosing Region Addresses and the Target Pathname

To begin, we choose the region addresses for the library's `.text` and `.data` sections from the segments reserved for private use on the 80386 Computer. Note that the region addresses must be on a segment boundary (4 MB):

```
.text    0x80600000
.data    0x80a00000
```

Also we choose the pathname for our target library:

```
/my/directory/libmaux_s
```

Selecting Library Contents

This example is for illustration purposes, and so we will include everything in the shared library. In a real case, it is unlikely that you would make a shared library with these three small routines, unless you had many programmers using them frequently.

Rewriting Existing Code

According to the guidelines given earlier in the chapter, we need to first minimize the global data. We realize that `total_calls`, `stats_logd`, and `stats_polyd` do not need to be visible outside the library, but are needed in multiple files within the library. Hence, we will use the `#hide` directive in our specification file to make these variables static after the shared library is built.

We need to define text and global data in separate source files. The only piece of global data we have left is `mauxerr`, which we will remove from `stats.c` and put in a new file `maux_defs.c`. We will also have to initialize it to zero, since shared libraries cannot have any uninitialized variables.

Next, we notice that there are some references to symbols that we do not define in our shared library (i.e. `log` and `malloc`). We can import these symbols. To do so, we create a new header file, `import.h`, which will be included in each of `log.c`, `poly.c`, and `stats.c`. The header file defines C preprocessor macros for these symbols to make transparent the use of indirection in the actual C source files.

We use the `_libmaux` prefixes on the pointers to the symbols because those pointers are made external, and the use of the library name as a prefix helps prevent name conflicts.

```
/* New header file import.h */
#define malloc (*_libmaux_malloc)
#define log (*_libmaux_log)

extern char * malloc();
extern double log();
```

Now, we need to define the imported symbol pointers somewhere. We have already created a file for global data `maux_defs.c`, so we will add the definitions to it.

```
/* Data file maux_defs.c */

int mauxerr = 0;
double (*_libmaux_log)() = 0;
char * (*_libmaux_malloc)() = 0;
```

14

Finally, we observe that there are floating-point operations in the code, and we remember that the routines for these cannot be imported. (If we tried to write the specification file and build the shared library without taking this into account, `mkshlib` would give us errors about unresolved references.) This means we will have to use the `#objects noload` directive in our specification file to search the C host shared library to resolve the references.

Building a Shared Library

Writing the Specification File

This is the specification file for **libmaux**:

Figure 14-9 Specification File

```
1  ##
2  ## libmaux.sl - libmaux specification lfile
3  #address .text 0x80680000
4  #address .data 0x806a0000
5  #target /my/directory/libmaux_s
6  #branch
7      polyd      1
8      logd       2
9      maux_stat  3
10 #objects
11     maux_defs.o
12     poly.o
13     log.o
14     stats.o
15 #objects noload
16     -lc s
17 #hide linker *
18 #export linker
19     mauxerr
20 #init maux_defs.o
21     _libmaux_mallocmalloc
22     _libmaux_loglog
```

Briefly, here is what the specification file does. Lines 1 and 2 are comment lines. Lines 3 and 4 give the virtual addresses for the shared library text and data regions, respectively. Line 5 gives the pathname of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Line 6 contains the **#branch** directive. Line 7 through 9 specify the branch table. They assign the functions **polyd()**, **logd()**, and **maux_stat()** to branch table entries 1, 2, and 3. Only external text symbols, such as C functions, should be placed in the branch table.

Line 10 contains the **#objects** directive. Lines 11 through 14 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, an addition of static data to **poly.o**, for example, would move external data symbols and break compatibility.

Line 15 contains the **#objects noload** directive, and line 16 gives information about where to resolve the references to the floating-point routines.

Lines 17 through 19 contain the **#hide linker** and **#export linker** directives, which tell what external symbols are to be left external after the shared library is built. Together, these **#hide** and **#export** directives say that only **mauxerr** will remain external. The symbols in the branch table and those specified in the **#init** directive will remain external by definition.

Line 20 contains the **#init** directive. Lines 21 and 22 give imported symbol information for the object file **maux_defs.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libmaux** will hold a pointer to **malloc**, and so on.

Building the Shared Library

Now, we have to compile the **.o** files as we would for any other library:

```
cc -c maux_defs.c poly.c log.c stats.c
```

Next, we need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib -s libmaux.sl -t libmaux_s -h libmaux_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libmaux_s.a**, and the target library, **libmaux_s**. Before any **a.out** files built with **libmaux_s.a** can be executed, the target shared library **libmaux_s** will have to be moved to **/my/directory/libmaux_s** as specified in the specification file.

Using the Shared Library

To use the shared library with a file, **x.c**, which contains a reference to one or more of the routines in **libmaux**, you would issue the following command line:

```
cc x.c libmaux_s.a -lm -lc_s
```

This command line causes the following:

- the imported symbol pointer reference to **log** is resolved from **libm**
- the imported symbol pointer reference to **malloc** is resolved with the shared version from **libc_s**.

The most important thing to note from the command line, however, is that you have to specify the C host shared library (in this case with the **-lc_s**) on the command line, since **libmaux** was built with direct references to the floating-point routines in that library.

Summary

This chapter describes the UNIX System shared libraries and explains how to use them. It also explains how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the UNIX System on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.

Chapter 15

Interprocess Communication

Introduction 15-1

Messages 15-2

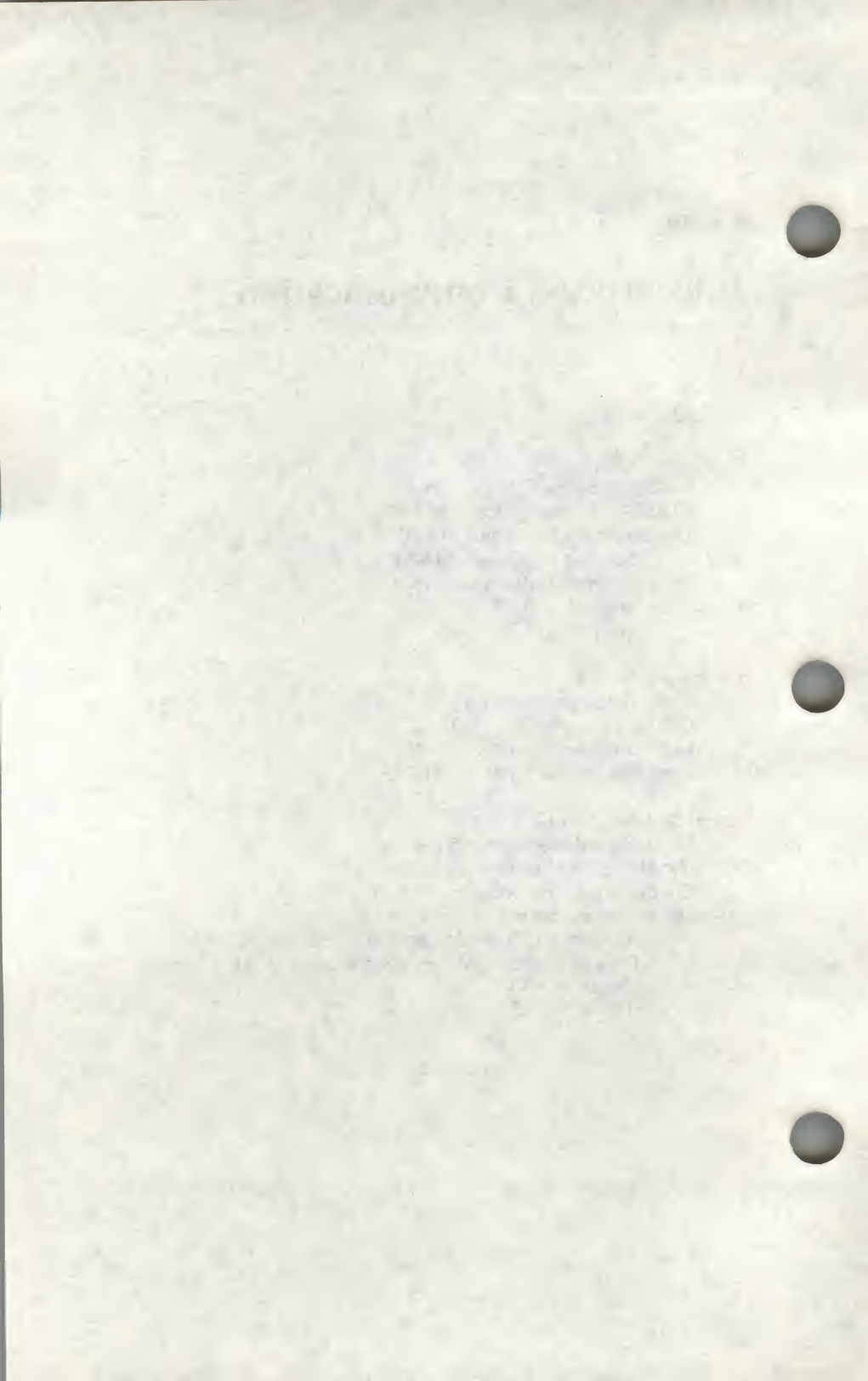
- Getting Message Queues 15-7
- Controlling Message Queues 15-13
- Operations for Messages 15-20
 - Sending a Message 15-21
 - Receiving Messages 15-22
 - msgsnd** 15-24
 - msgrcv** 15-25

Semaphores 15-32

- Using Semaphores 15-34
- Getting Semaphores 15-37
- Controlling Semaphores 15-45
- Operations on Semaphores 15-55

Shared Memory 15-62

- Using Shared Memory 15-63
- Getting Shared Memory Segments 15-67
- Controlling Shared Memory 15-73
- Operations for Shared Memory 15-81
 - Attaching a Shared Memory Segment 15-82
 - Detaching Shared Memory Segments 15-83
 - shmat** 15-84
 - shmdt** 15-85



Introduction

The UNIX System supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC. Included are several example programs that show the use of the IPC system calls.

Since there are many ways in the C Programming Language to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger program that makes use of a particular function that the calls provide.

Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations:

- sending
- receiving

Before a message can be sent or received by a process, a process must have the UNIX System generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget(S)** system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl(S)** system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl()** to perform various other control functions.

Processes which have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process which is attempting to send a message can wait until the process which is to receive the message is ready and vice versa. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- The process is successful in sending or receiving its message.
- The process receives a signal.
- The message facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (-1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store (header) information about each message that is being sent or received. This information includes the following for each message:

- pointer to the next message on queue
- message type
- message text size
- message text address

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue
- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender

Messages

- PID of last message receiver
- last message send time
- last message receive time
- last change time

Note

All include files discussed in this chapter are located in the `/usr/include` or `/usr/include/sys` directories.

The C Programming Language data structure definition for message information contained in the message queue is located in the header file `#include <sys/msg.h>` and is as follows:

```
15 struct msg
{
    struct msg *msg_next; /* ptr to next message on q */
    long      msg_type;   /* message type */
    short     msg_ts;     /* message text size */
    short     msg_spot;   /* message text map address */
};
```

Likewise, the structure definition for the associated data structure is located in the `#include <sys/msg.h>` header file and is as follows:

```
struct msqid_ds
{
    struct ipc_perm msg_perm; /* operation permission struct */
    struct msg *msg_first; /* ptr to first message on q */
    struct msg *msg_last; /* ptr to last message on q */
    ushort msg_cbytes; /* current # bytes on q */
    ushort msg_qnum; /* # of messages on q */
    ushort msg_qbytes; /* max # of bytes on q */
    ushort msg_lspid; /* pid of last msgsnd */
    ushort msg_lrpid; /* pid of last msgrcv */
    time_t msg_stime; /* last msgsnd time */
    time_t msg_rtime; /* last msgrcv time */
    time_t msg_ctime; /* last change time */
};
```

Note that the `msg_perm` member of this structure uses `ipc_perm` as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The definition of the **ipc_perm** data structure is located in the header file **#include <sys/ipc.h>** and is as follows:

Figure 15-1 ipc_perm Data Structure

```
struct ipc_perm
{
    ushortuid; /* owner's user id */
    ushortgid; /* owner's group id */
    ushortcuid; /* creator's user id */
    ushortcgid; /* creator's group id */
    ushortmode; /* access modes */
    ushortseq; /* slot usage sequence number */
    key_t key; /* key */
};
```

The structure is common for all IPC facilities.

The **msgget(S)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **msgflg** argument that it receives:

- to get a new **msqid** and create an associated message queue and data structure for it
- to return an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget()** system call. For the first task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**. This occurs provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (**IPC_PRIVATE = 0**). When specified, a new **msqid** is always returned with an associated message queue and data structure created for it, unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, for security reasons the **KEY** field for the **msqid** is all zeros.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not desire to have an existing **msqid** returned, a control command (**IPC_EXCL**) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in the "Using **msgget**" section of this chapter.

Messages

When performing the first task, the process which calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the "Controlling Message Queues" section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations [**msgop()**] and message control [**msgctl()**] can be used.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd()** and **msgrcv()**. Refer to the "Operations for Messages" section in this chapter for details of these system calls.

Message control is done by using the **msgctl(S)** system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (**msqid**)
- to change operation permissions for a message queue
- to change the size (**msg_qbytes**) of the message queue for a particular **msqid**
- to remove a particular **msqid** from the UNIX Operating System along with its associated message queue and data structure

Refer to the "Controlling Message Queues" section in this chapter for details of the **msgctl()** system call.

Getting Message Queues

This section gives a detailed description of using the **msgget(S)** system call along with an example program illustrating its use.

Using msgget

The synopsis found in the **msgget(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX Operating System.

The following line in the synopsis informs you that **msgget()** is a function with two formal arguments that returns an integer type value upon successful completion (**msqid**).

```
int msgget (key, msgflg)
```

The next two lines declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file.

```
key_t key;
int msgflg;
```

The integer returned from this function upon successful completion is the message queue identifier (**msqid**) that was discussed earlier.

As declared, the process calling the **msgget()** system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

Messages

A new **msqid** with an associated message queue and data structure is provided if one of the following conditions exists:

- **key** is equal to **IPC_PRIVATE**
- **key** is passed a unique integer, and **msgflg** ANDed with **IPC_CREAT** is **TRUE**.

The value passed to the **msgflg** argument must be an integer; an octal value will specify the following and be easier to understand:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes, and execution modes determine the user/group/other attributes of the **msgflg** argument. They are collectively referred to as "operation permissions." Figure 15-2 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Figure 15-2 Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **msg.h** header file which can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). Figure 15-3 contains the names of the constants which apply to the `msgget()` system call along with their values. They are also referred to as flags and are defined in the `ipc.h` header file.

Figure 15-3 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for the `msgflg` argument is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; bit positions and values for the control commands in relation to those of the operation permissions make this possible. An example of determining the `msgflg` argument follows.

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
Read by User	=	0 0 4 0 0	0 000 000 100 000 000
msgflg	=	0 1 4 0 0	0 000 001 100 000 000

The `msgflg` value can easily be set by using the names of the flags in conjunction with the octal operation permissions value:

```
msqid = msgget (key, (IPC_CREAT | 0400));
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the `msgget(S)` page in the *Programmer's Reference*, success or failure of this system call depends upon the argument values for `key` and `msgflg` or system tunable parameters. The system call will attempt to return a new `msqid` if one of the following conditions exists:

- `key` is equal to `IPC_PRIVATE` (0)
- `key` does not already have a `msqid` associated with it, and `(msgflg & IPC_CREAT)` is TRUE (not zero).

Messages

The **key** argument can be set to `IPC_PRIVATE` in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

or

```
msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the `MSGMNI` system tunable parameter always causes a failure. The `MSGMNI` system tunable parameter determines the maximum number of unique message queues (**msqid**'s) in the UNIX Operating System.

The second condition is satisfied if the value for **key** is not already associated with a **msqid** and the bitwise ANDing of **msgflg** and `IPC_CREAT` is TRUE (not zero). This means that the **key** is unique (not in use) within the UNIX Operating System for this facility type and that the `IPC_CREAT` flag is set (**msgflg** | `IPC_CREAT`). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
msgflg = x 1 x x x    (x = 0 or 1)
& IPC_CREAT = 0 1 0 0 0
result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or TRUE.

`IPC_EXCL` is another control command used in conjunction with `IPC_CREAT` to exclusively have the system call fail if, and only if, a **msqid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msqid** when it has not. In other words, when both `IPC_CREAT` and `IPC_EXCL` are specified, a new **msqid** is returned if the system call is successful which is when `[(msgflg & IPC_CREAT) & (msgflg & IPC_EXCL)]` is false (0).

Refer to the `msgget(S)` page in the *Programmer's Reference* for specific, associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 15-4) is a menu-driven program which allows all possible combinations of using the `msgget(S)` system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **msgget(S)** entry in the *Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purposes are as follows:

- **key**—is used to pass the value for the desired **key**.
- **opperm**—is used to store the desired operation permissions.
- **flags**—is used to store the desired control commands (flags).
- **opperm_flags**—is used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument.
- **msqid**—is used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

The system call is made next, and the result is stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 55). If **msqid** equals -1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57 and 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

Messages

The example program for the **msgget(S)** system call follows. It is suggested that the source program file be named **msgget.c** and that the executable file be named **msgget**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed, it will fail.

Figure 15-4 msgget() System Call Example (Sheet 1 of 2)

```
1  /*This is a program to illustrate
2  **the message get, msgget(),
3  **system call capabilities.*/

4  #include    <stdio.h>
5  #include    <sys/types.h>
6  #include    <sys/ipc.h>
7  #include    <sys/msg.h>
8  #include    <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key;          /*declare as long integer*/
13     int opperm, flags;
14     int msqid, opperm_flags;
15     /*Enter the desired key*/
16     printf("Enter the desired key in hex = ");
17     scanf("%x", &key);

18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags                = 0\n");
27     printf("IPC_CREAT                    = 1\n");
28     printf("IPC_EXCL                     = 2\n");
29     printf("IPC_CREAT and IPC_EXCL        = 3\n");
30     printf("Flags                          = ");

31     /*Get the flag(s) to be set.*/
32     scanf("%d", &flags);
```

Figure 15-4 msgget() System Call Example (Sheet 2 of 2)

```

33      /*Check the values.*/
34      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35             key, opperm, flags);
36      /*Incorporate the control fields (flags) with
37       the operation permissions*/
38      switch (flags)
39      {
40      case 0:      /*No flags are to be set.*/
41                  opperm_flags = (opperm | 0);
42                  break;
43      case 1:      /*Set the IPC_CREAT flag.*/
44                  opperm_flags = (opperm | IPC_CREAT);
45                  break;
46      case 2:      /*Set the IPC_EXCL flag.*/
47                  opperm_flags = (opperm | IPC_EXCL);
48                  break;
49      case 3:      /*Set the IPC_CREAT and IPC_EXCL flags.*/
50                  opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51      }

52      /*Call the msgget system call.*/
53      msqid = msgget (key, opperm_flags);

54      /*Perform the following if the call is unsuccessful.*/
55      if (msqid == -1)
56      {
57          printf ("\nThe msgget system call failed!\n");
58          printf ("The error number = %d\n", errno);
59      }

60      /*Return the msqid upon successful completion.*/
61      else
62          printf ("\nThe msqid = %d\n", msqid);
63      exit(0);
64  }

```

15

Controlling Message Queues

This section gives a detailed description of using the **msgctl** system call. It also provides an example program which allows all of its capabilities to be exercised.

Messages

Using msgctl

The synopsis found in the **msgctl(3)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The **msgctl()** system call requires three arguments to be passed to it, and it returns an integer value. Upon successful completion, a zero value is returned. When unsuccessful, a -1 is returned.

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

- IPC_STAT** returns the status information contained in the associated data structure for the specified **msqid**, and places it in the data structure pointed to by the ***buf** pointer in the user memory area.
- IPC_SET** for the specified **msqid**, sets the effective user and group identification, operation permissions, and the number of bytes for the message queue.
- IPC_RMID** removes the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an **IPC_SET** or **IPC_RMID** control command. Read permission is required to perform the **IPC_STAT** control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-5) is a menu-driven program which allows all possible combinations of using the **msgctl(S)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgctl(S)** entry in the *Programmer's Reference*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purpose are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **bytes**—used to store the IPC_SET value for the number of bytes in the message queue (**msg_qbytes**)
- **rtrn**—used to store the return integer value from the system call
- **msqid**—used to store and pass the message queue identifier to the system call
- **command**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member is to be changed for the IPC_SET control command
- **msqid_ds**—used to receive the specified message queue identifier's data structure when an IPC_STAT control command is performed

Messages

- ***buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the **msg.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that, although the ***buf** pointer is declared to be a pointer to a data structure of the **msqid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, the following text explains how it works.

First, the program prompts for a valid message queue identifier which is stored at the address of the **msqid** variable (lines 19 and 20). This is required for every **msgctl** system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37 and 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 108 and 109). If the system call is successful, a message indicates this, along with the message queue identifier used (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the UNIX Operating System. Note that the ***buf** pointer is not required as an argument to perform this control command, and its value can be zero or `NULL`. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the **msgctl()** system call follows. It is suggested that the source program file be named **msgctl.c** and that the executable file be named **msgctl**.

When compiling C programs that use floating point operations, the **-f** option should be used on the `cc` command line. If this option is not used, the program will compile successfully, but when the program is executed, it will fail.

Figure 15-5 msgctl() System Call Example (Sheet 1 of 3)

```

1  /*This is a program to illustrate
2  **the message control, msgctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode, bytes;
15     int rtm, msgid, command, choice;
16     struct msgid_ds msgid_ds, *buf;
17     buf = &msgid_ds;

18     /*Get the msgid, and command.*/
19     printf("Enter the msgid = ");
20     scanf("%d", &msgid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");
23     printf("IPC_STAT   = 1\n");
24     printf("IPC_SET    = 2\n");
25     printf("IPC_RMID   = 3\n");
26     printf("Entry      = ");
27     scanf("%d", &command);

28     /*Check the values.*/
29     printf("\nmsgid = %d, command = %d\n",
30           msgid, command);

31     switch (command)
32     {
33     case 1: /*Use msgctl() to duplicate
34             the data structure for
35             msgid in the msgid_ds area pointed
36             to by buf and then print it out.*/
37         rtm = msgctl(msgid, IPC_STAT,
38                     buf);
39         printf("\nThe USER ID = %d\n",
40               buf->msg_perm.uid);
41         printf("The GROUP ID = %d\n",
42               buf->msg_perm.gid);
43         printf("The operation permissions = 0%o\n",
44               buf->msg_perm.mode);
45         printf("The msg_qbytes = %d\n",
46               buf->msg_qbytes);
47         break;

```

Figure 15-5 msgctl() System Call Example (Sheet 2 of 3)

```

48     case 2:      /*Select and change the desired
49                  member(s) of the data structure.*/
50                  /*Get the original data for this msgid
51                     data structure first.*/
52     rtm = msgctl(msgid, IPC_STAT, buf);
53     printf("\nEnter the number for the\n");
54     printf("member to be changed:\n");
55     printf("msg_perm.uid   = 1\n");
56     printf("msg_perm.gid   = 2\n");
57     printf("msg_perm.mode   = 3\n");
58     printf("msg_qbytes     = 4\n");
59     printf("Entry          = ");

60     scanf("%d", &choice);
61     /*Only one choice is allowed per
62        pass as an illegal entry will
63        cause repetitive failures until
64        msgid_ds is updated with
65        IPC_STAT.*/

66     switch(choice) {
67     case 1:
68         printf("\nEnter USER ID = ");
69         scanf("%d", &uid);
70         buf->msg_perm.uid = uid;
71         printf("\nUSER ID = %d\n",
72                buf->msg_perm.uid);
73         break;
74     case 2:
75         printf("\nEnter GROUP ID = ");
76         scanf("%d", &gid);
77         buf->msg_perm.gid = gid;
78         printf("\nGROUP ID = %d\n",
79                buf->msg_perm.gid);
80         break;
81     case 3:
82         printf("\nEnter MODE = ");
83         scanf("%o", &mode);
84         buf->msg_perm.mode = mode;
85         printf("\nMODE = %o\n",
86                buf->msg_perm.mode);
87         break;
88     case 4:
89         printf("\nEnter msg_bytes = ");
90         scanf("%d", &bytes);
91         buf->msg_qbytes = bytes;
92         printf("\nmsg_qbytes = %d\n",
93                buf->msg_qbytes);
94         break;
95     }

```


Figure 15-5 msgctl() System Call Example (Sheet 3 of 3)

```

96         /*Do the change.*/
97         rtm = msgctl(msqid, IPC_SET,
98             buf);
99         break;

100     case 3:    /*Remove the msqid along with its
101                associated message queue
102                and data structure.*/
103         rtm = msgctl(msqid, IPC_RMID, NULL);
104     }
105     /*Perform the following if the call is unsuccessful.*/
106     if(rtm == -1)
107     {
108         printf ("\nThe msgctl system call failed!\n");
109         printf ("The error number = %d\n", errno);
110     }
111     /*Return the msqid upon successful completion.*/
112     else
113         printf ("\nMsgctl was successful for msqid = %d\n",
114             msqid);
115     exit (0);
116 }
```

Operations for Messages

This section gives a detailed description of using the **msgsnd(S)** and **msgrcv(S)** system calls, along with an example program which allows all of their capabilities to be exercised.

Using msgop

The synopsis found in the **msgop(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it, and it returns an integer value. Upon successful completion, a zero value is returned. When unsuccessful, a -1 is returned.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget()** system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system tunable parameter.

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl()** IPC_SET control command, but only the super-user can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg & IPC_NOWAIT = 0**); this would occur if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a -1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Receiving Messages

The **msgrcv()** system call requires five arguments to be passed to it, and it returns an integer value. Upon successful completion, a value equal to the number of bytes received is returned. When unsuccessful, a -1 is returned.

The **msqid** argument must be a valid, non-negative, integer value; that is, it must have already been created by using the **msgget()** system call.

The **msgp** argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The **msgsz** argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the **msgflg** argument.

The **msgtyp** argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The **msgflg** argument allows the "blocking message operation" to be performed if the **IPC_NOWAIT** flag is not set (**msgflg & IPC_NOWAIT = 0**); this would occur if there is not a message on the message queue of the desired type (**msgtyp**) to be received. If the **IPC_NOWAIT** flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. **msgflg** can also specify that the system call fail if the message is longer than the **size** to be received; this is done by not setting the **MSG_NOERROR** flag in the **msgflg** argument (**msgflg & MSG_NOERROR = 0**). If the **MSG_NOERROR** flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv()**.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-6) is a menu-driven program which allows all possible combinations of using the **msgsnd()** and **msgrcv(S)** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgop(S)** entry in the *Programmer's Reference*. Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purposes are:

- **sndbuf**—is used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13). The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message **size** (MSGMAX) for your computer, where in **msgbuf** it is set to one (C) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.
- **rcvbuf**—is used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13).
- ***msgp**—is used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers.
- **i**—is used as a counter to input characters from the keyboard, to store them in the array, and to keep track of the message length for the **msgsnd()** system call; it is also used as a counter to output the received message for the **msgrcv()** system call.
- **c**—is used to receive the input character from the **getchar()** function (line 50).
- **flag**—is used to store the code of IPC_NOWAIT for the **msgsnd()** system call (line 61).

Messages

- **flags**—is used to store the code of the `IPC_NOWAIT` or `MSG_NOERROR` flags for the `msgrcv()` system call (line 117).
- **choice**—is used to store the code for sending or receiving (line 30).
- **rtrn**—is used to store the return values from all system calls.
- **msqid**—is used to store and pass the desired message queue identifier for both system calls.
- **msgsz**—is used to store and pass the size of the message to be sent or received.
- **msgflg**—is used to pass the value of flag for sending or the value of flags for receiving.
- **msgtyp**—is used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the `msgctl()` (`IPC_STAT`) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd

When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored at the address of the variable **msgtyp** (line 42), and then (line 43) it is put into the **mtyp** member of the data structure pointed to by **msgp**.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the **mtext** array of the data structure (lines 48-51). This will continue until an end of file is recognized, which for the `getchar()` function is a `<CTL>D` immediately following a carriage return (`<Return>`). When this happens, the **size** of the message is determined by adding one to the **i** counter (lines 52 and 53), as it stored the message beginning in the zero array element of **mtext**. Keep in

mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of **msgsz**.

The message is immediately echoed from the **mtext** array of the **sndbuf** data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the **IPC_NOWAIT** flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, **IPC_NOWAIT** is logically ORed with **msgflg**; otherwise, **msgflg** is set to zero.

The **msgsnd()** system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value is printed, which should be zero (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure which are updated. They are described as follows:

msg_qnum	represents the total number of messages on the message queue; it is incremented by one.
msg_lspid	contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.
msg_stime	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the message queue from which you will receive the message is requested, and it is stored at the address of **msqid** (lines 100-103).

Messages

The message type is requested, and it is stored at the address of **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending upon the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of **msgsz** (lines 134-137).

The **msgrcv()** system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, there are three members of the associated data structure which are updated; they are described as follows:

msg_qnum	contains the number of messages on the message queue; it is decremented by one.
msg_lrpId	contains the process identification (PID) of the last process receiving a message; it is set accordingly.
msg_rtime	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the **msgop()** system calls follows. It is suggested that the program be put into a source file called **msgop.c** and then into an executable file called **msgop**.

Figure 15-6 msgop() System Call Example (Sheet 1 of 5)

```

1  /*This is a program to illustrate
2  **the message operations, msgop(),
3  **system call capabilities.
4  */
5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>
10 struct msgbuf {
11     long mtype;
12     char mtext[8192];
13 } sndbuf, rcvbuf, *msgp;
14 /*Start of main C language program*/
15 main()
16 {
17     extern int errno;
18     int i, c, flag, flags, choice;
19     int rtm, msqid, msgsz, msgflg;
20     long mtype, msgtyp;
21     struct msqid_ds msqid_ds, *buf;
22     buf = &msqid_ds;
23     /*Select the desired operation.*/
24     printf("Enter the corresponding\n");
25     printf("code to send or\n");
26     printf("receive a message:\n");
27     printf("Send          = 1\n");
28     printf("Receive         = 2\n");
29     printf("Entry           = ");
30     scanf("%d", &choice);
31     if(choice == 1) /*Send a message.*/
32     {
33         msgp = &sndbuf; /*Point to user send structure.*/
34         printf("\nEnter the msqid of\n");
35         printf("the message queue to\n");
36         printf("handle the message = ");
37         scanf("%d", &msqid);
38         /*Set the message type.*/
39         printf("\nEnter a positive integer\n");
40         printf("message type (long) for the\n");
41         printf("message = ");
42         scanf("%d", &msgtyp);
43         msgp->mtype = msgtyp;
44         /*Enter the message to send.*/
45         printf("\nEnter a message: \n");

```

Figure 15-6 msgop() System Call Example (Sheet 2 of 5)

```
46      /*A control-d (^d) terminates as
47      EOF.*/

48      /*Get each character of the message
49      and put it in the mtext array.*/
50      for(i = 0; ((c = getchar()) != EOF); i++)
51          sndbuf.mtext[i] = c;

52      /*Determine the message size.*/
53      msgsz = i + 1;

54      /*Echo the message to send.*/
55      for(i = 0; i < msgsz; i++)
56          putchar(sndbuf.mtext[i]);

57      /*Set the IPC_NOWAIT flag if
58      desired.*/
59      printf("\nEnter a 1 if you want the\n");
60      printf("the IPC_NOWAIT flag set: ");
61      scanf("%d", &flag);
62      if(flag == 1)
63          msgflg |= IPC_NOWAIT;
64      else
65          msgflg = 0;

66      /*Check the msgflg.*/
67      printf("\nmsgflg = 0%o\n", msgflg);

68      /*Send the message.*/
69      rtm = msgsnd(msgqid, msgp, msgsz, msgflg);
70      if(rtm == -1)
71          printf("\nMsgsnd failed. Error = %d\n",
72                errno);
73      else {
74          /*Print the value of test which
75          should be zero for successful.*/
76          printf("\nValue returned = %d\n", rtm);
77          /*Print the size of the message
78          sent.*/
79          printf("\nMsgsz = %d\n", msgsz);

80          /*Check the data structure update.*/
81          msgctl(msgqid, IPC_STAT, buf);
```


Figure 15-6 msgop() System Call Example (Sheet 3 of 5)

```

82         /*Print out the affected members.*/
83
84         /*Print the incremented number of
85         messages on the queue.*/
86         printf("\nThe msg_qnum = %d\n",
87             buf->msg_qnum);
88         /*Print the process id of the last sender.*/
89         printf("The msg_lspid = %d\n",
90             buf->msg_lspid);
91         /*Print the last send time.*/
92         printf("The msg_stime = %d\n",
93             buf->msg_stime);
94     }
95
96     if(choice == 2) /*Receive a message.*/
97     {
98         /*Initialize the message pointer
99         to the receive buffer.*/
100         msgp = &rcvbuf;
101
102         /*Specify the message queue which contains
103         the desired message.*/
104         printf("\nEnter the msgqid = ");
105         scanf("%d", &msgqid);
106
107         /*Specify the specific message on the queue
108         by using its type.*/
109         printf("\nEnter the msgtyp = ");
110         scanf("%d", &msgtyp);
111
112         /*Configure the control flags for the
113         desired actions.*/
114         printf("\nEnter the corresponding code\n");
115         printf("to select the desired flags: \n");
116         printf("No flags                = 0\n");
117         printf("MSG_NOERROR                = 1\n");
118         printf("IPC_NOWAIT                = 2\n");
119         printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
120         printf("Flags                = ");
121         scanf("%d", &flags);

```

Figure 15-6 msgop() System Call Example (Sheet 4 of 5)

```

118         switch(flags) {
119             /*Set msgflg by ORing it with the appropriate
120                flags (constants).*/
121         case 0:
122             msgflg = 0;
123             break;
124         case 1:
125             msgflg |= MSG_NOERROR;
126             break;
127         case 2:
128             msgflg |= IPC_NOWAIT;
129             break;
130         case 3:
131             msgflg |= MSG_NOERROR | IPC_NOWAIT;
132             break;
133         }

134         /*Specify the number of bytes to receive.*/
135         printf("\nEnter the number of bytes\n");
136         printf("to receive (msgsz) = ");
137         scanf("%d", &msgsz);

138         /*Check the values for the arguments.*/
139         printf("\nmsgid = %d\n", msgid);
140         printf("\nmsgtyp = %d\n", msgtyp);
141         printf("\nmsgsz = %d\n", msgsz);
142         printf("\nmsgflg = 0x%x\n", msgflg);

143         /*Call msgrcv to receive the message.*/
144         rtm = msgrcv(msgid, msgp, msgsz, msgtyp, msgflg);

145         if(rtm == -1) {
146             printf("\nMsgrcv failed. ");
147             printf("Error = %d\n", errno);
148         }
149         else {
150             printf("\nMsgctl was successful\n");
151             printf("for msgid = %d\n",
152                 msgid);

153             /*Print the number of bytes received,
154                it is equal to the return
155                value.*/
156             printf("Bytes received = %d\n", rtm);

157             /*Print the received message.*/
158             for(i = 0; i<=rtm; i++)
159                 putchar(rcvbuf.mtext[i]);
160         }

```

Figure 15-6 msgop() System Call Example (Sheet 5 of 5)

```
161      /*Check the associated data structure.*/
162      msgctl(msgid, IPC_STAT, buf);
163      /*Print the decremented number of messages.*/
164      printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165      /*Print the process id of the last receiver.*/
166      printf("The msg_lpid = %d\n", buf->msg_lpid);
167      /*Print the last message receive time*/
168      printf("The msg_rtime = %d\n", buf->msg_rtime);
169  }
170 }
```

Semaphores

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the UNIX Operating System has the ability to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, SEMMSL has a default value of 25. Semaphore sets are created by using the **semget(S)** system call.

The process performing the **semget(S)** system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl()**, semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl()** to perform other control functions.

15 Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop(S)** system call (which is documented in the *Programmer's Reference*):

- incremented
- decremented

To increment a semaphore, a positive integer value of the desired magnitude is passed to the **semop(S)** system call. To decrement a semaphore, a negative (-) value of the desired magnitude is passed.

The UNIX Operating System ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the **semop(S)** system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, the process is returned a known error code (-1), and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX Operating System until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop(S)**, semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed, or one "nonblocking operation" is unsuccessful and none are changed. All of this is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX Operating System to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation which is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Semaphores

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is user-selectable. The following members are in each structure within a semaphore set:

- semaphore text map address
- process identification (PID) performing last operation
- number of processes awaiting the semaphore value to become greater than its current value
- number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C Programming Language data structure definition for the semaphore set (array member) is located in the `#include <sys/sem.h>` header file and is as follows:

```
struct sem
{
    ushort  semval;          /* semaphore value */
    short   sempid;          /* pid of last operation */
    ushort  semncnt;         /* # awaiting semval > current value */
    ushort  semzcnt;         /* # awaiting semval = 0 */
};
```

Likewise, the structure definition for the associated semaphore data structure is also located in the `#include <sys/sem.h>` header file and is as follows:

```
struct semid_ds
{
    struct ipc_perm sem_perm; /* operation permission struct */
    struct sem      *sem_base; /* ptr to first semaphore in set */
    ushort          sem_nsems; /* # of semaphores in set */
    time_t          sem_otime; /* last semop time */
    time_t          sem_ctime; /* last change time */
};
```

15

Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the `#include <sys/ipc.h>` header file. It is shown in the "Messages" section.

The **semget(S)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **semflg** argument that it receives:

- to get a new **semid** and create an associated data structure and semaphore set for it
- to return an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget(S)** system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it, provided no system tunable parameter would be exceeded.

Semaphores

There is also a provision for specifying a **key** of value zero (0), which is known as the private **key** (`IPC_PRIVATE = 0`); when specified, a new **semid** is always returned with an associated data structure and semaphore set created for it, unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **semid** is all zeros.

When performing the first task, the process which calls **semget()** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Semaphores" section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned, a control command (`IPC_EXCL`) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section of this chapter.

15

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop(S)**] and semaphore control [**semctl()**] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop()**. Refer to the "Operations on Semaphores" section in this chapter for details of this system call.

Semaphore control is done by using the **semctl(S)** system call. These control operations permit you to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero

- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values, status, of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular **semid** from the UNIX Operating System along with its associated data structure and semaphore set.

Refer to the "Controlling Semaphores" section in this chapter for details of the **semctl(S)** system call.

Getting Semaphores

This section contains a detailed description of using the **semget(S)** system call along with an example program illustrating its use.

Using semget

The synopsis found in the **semget(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semg)
key_t key;
int nsems, semg;
```

The following line in the synopsis informs you that **semget()** is a function with three formal arguments that returns an integer type value upon successful completion (**semid**).

```
int semget (key, nsems, semflg)
```

The next two lines declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

```
key_t key;
int nsems, semflg;
```


The integer returned from this system call upon successful completion is the semaphore set identifier (**semid**) that was discussed earlier.

As declared, the process calling the **semget()** system call must supply three arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if one of the following conditions exists:

- **key** is equal to **IPC_PRIVATE**
- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with **IPC_CREAT** is **TRUE**.

The value passed to the **semflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes, and execution modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as "operation permissions." Figure 15-7 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

Figure 15-7 Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define'd** in the **sem.h** header file which can be used for the user (OWNER). They are:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Figure 15-8 contains the names of the constants which apply to the **semget(S)** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Figure 15-8 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for the **semflg** argument is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (!) them with the operation permissions; bit positions and values for the control commands in relation to those of the operation permissions make this possible. An example of determining the **semflg** argument follows.

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
CWI Read by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

The **semflg** value can easily be set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget(S)** entry in the *Programmer's Reference*, success or failure of this system call depends upon the actual argument values for **key**, **nsems**, **semflg** or system tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions exists:

- **key** is equal to **IPC_PRIVATE** (0)
- **key** does not already have a **semid** associated with it, and (**semflg** & **IPC_CREAT**) is TRUE (not zero).

The **key** argument can be set to `IPC_PRIVATE` in the following ways:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget ( 0, nsems, semflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the `SEMMNI`, `SEMMNS`, or `SEMMSL` system-tunable parameters will always cause a failure. The `SEMMNI` system-tunable parameter determines the maximum number of unique semaphore sets (**semid**'s) in the UNIX Operating System. The `SEMMNS` system-tunable parameter determines the maximum number of semaphores in all semaphore sets systemwide. The `SEMMSL` system-tunable parameter determines the maximum number of semaphores in each semaphore set.

The second condition is satisfied if the value for **key** is not already associated with a **semid** and the bitwise ANDing of **semflg** and `IPC_CREAT` is TRUE (not zero). This means that the **key** is unique (not in use) within the UNIX Operating System for this facility type and that the `IPC_CREAT` flag is set (**semflg** | `IPC_CREAT`).

The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
semflg = x 1 x x x    (x = 0 or 1)
& IPC_CREAT = 0 1 0 0 0
result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or TRUE. `SEMMNI`, `SEMMNS`, and `SEMMSL` apply here also, just as for condition one.

`IPC_EXCL` is another control command used in conjunction with `IPC_CREAT` to exclusively have the system call fail if, and only if, a **semid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both `IPC_CREAT` and `IPC_EXCL` are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the **key** equals zero (`IPC_PRIVATE`) and no system-tunable parameters are exceeded.

Refer to the `semget(S)` manual page for specific, associated data structure initialization for successful completion.

Example Program

The example program in this section (Figure 15-9) is a menu-driven program which allows all possible combinations of using the **semget(S)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **semget(S)** entry in the *Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purpose are as follows:

- **key**—is used to pass the value for the desired **key**.
- **opperm**—is used to store the desired operation permissions.
- **flags**—is used to store the desired control commands (flags).
- **opperm_flags**—is used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument.
- **semid**—is used for returning the semaphore set identification number for a successful system call or the error code (-1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

Semaphores

The system call is made next, and the result is stored at the address of the **semid** variable (lines 60 and 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (-1), it is tested to see if an error occurred (line 63). If **semid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65 and 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget(S)** system call follows. It is suggested that the source program file be named **semget.c** and that the executable file be named **semget**.

Figure 15-9 semget() System Call Example (Sheet 1 of 2)

```

1  /*This is a program to illustrate
2  **the semaphore get, semget(),
3  **system call capabilities.*/

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10 main()
11 {
12     key_t key; /*declare as long integer*/
13     int opperm, flags, nsms;
14     int semid, opperm_flags;

15     /*Enter the desired key*/
16     printf("\nEnter the desired key in hex = ");
17     scanf("%x", &key);

18     /*Enter the desired octal operation
19     permissions.*/
20     printf("\nEnter the operation\n");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);

23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags           = 0\n");
27     printf("IPC_CREAT                 = 1\n");
28     printf("IPC_EXCL                   = 2\n");
29     printf("IPC_CREAT and IPC_EXCL    = 3\n");
30     printf("Flags                     = ");

31     /*Get the flags to be set.*/
32     scanf("%d", &flags);

```


Figure 15-9 `semget()` System Call Example (Sheet 2 of 2)

```

33      /*Error checking (debugging)*/
34      printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35              key, opperm, flags);
36      /*Incorporate the control fields (flags) with
37         the operation permissions.*/
38      switch (flags)
39      {
40      case 0:      /*No flags are to be set.*/
41          opperm_flags = (opperm | 0);
42          break;
43      case 1:      /*Set the IPC_CREAT flag.*/
44          opperm_flags = (opperm | IPC_CREAT);
45          break;
46      case 2:      /*Set the IPC_EXCL flag.*/
47          opperm_flags = (opperm | IPC_EXCL);
48          break;
49      case 3: /*Set the IPC_CREAT and IPC_EXCL
50              flags.*/
51          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52      }

53      /*Get the number of semaphores for this set.*/
54      printf ("\nEnter the number of\n");
55      printf ("desired semaphores for\n");
56      printf ("this set (25 max) = ");
57      scanf ("%d", &nsens);

58      /*Check the entry.*/
59      printf ("\nNsens = %d\n", nsens);

60      /*Call the semget system call.*/
61      semid = semget (key, nsens, opperm_flags);

62      /*Perform the following if the call is unsuccessful.*/
63      if (semid == -1)
64      {
65          printf ("The semget system call failed!\n");
66          printf ("The error number = %d\n", errno);
67      }
68      /*Return the semid upon successful completion.*/
69      else
70          printf ("\nThe semid = %d\n", semid);
71      exit (0);
72  }

```

Controlling Semaphores

This section contains a detailed description of using the **semctl(S)** system call along with an example program which allows all of its capabilities to be exercised.

Using semctl

The synopsis found in the **semctl(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
    int val;
    struct semid_ds *bu;
    ushort array[];
} arg;
```

The **semctl(S)** system call requires four arguments to be passed to it, and it returns an integer value.

The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget(S)** system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following control commands (flags):

- **GETVAL**—returns the value of a single semaphore within a semaphore set.
- **SETVAL**—sets the value of a single semaphore within a semaphore set.
- **GETPID**—returns the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set.

Semaphores

- **GETNCNT**—returns the number of processes waiting for the value of a particular semaphore to become greater than its current value.
- **GETZCNT**—returns the number of processes waiting for the value of a particular semaphore to be equal to zero.
- **GETALL**—returns the values for all semaphores in a semaphore set.
- **SETALL**—sets all semaphore values in a semaphore set.
- **IPC_STAT**—returns the status information contained in the associated data structure for the specified **semid**, and places it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**.
- **IPC_SET**—for the specified semaphore set (**semid**), sets the effective user/group identification and operation permissions.
- **IPC_RMID**—removes the specified (**semid**) semaphore set along with its associated data structure.

15 A process must have an effective user identification of **OWNER/CREATOR** or super-user to perform an **IPC_SET** or **IPC_RMID** control command. Read/alter permission is required as applicable for the other control commands.

The **arg** argument is used to pass the system call the appropriate union member for the control command to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **semget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-10) is a menu-driven program which allows all possible combinations of using the **semctl(S)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **semctl(S)** entry in the *Programmer's Reference*. Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purpose are as follows:

- **semid_ds**—is used to receive the specified semaphore set identifier's data structure when an **IPC_STAT** control command is performed.
- **c**—is used to receive the input values from the **scanf(S)** function (line 117) when performing a **SETALL** control command.
- **i**—is used as a counter to increment through the union **arg.array** when displaying the semaphore values for a **GETALL** (lines 97-99) control command, and when initializing the **arg.array** when performing a **SETALL** (lines 115-119) control command.
- **length**—is used as a variable to test for the number of semaphores in a set against the **i** counter variable (lines 97 and 115).
- **uid**—is used to store the **IPC_SET** value for the effective user identification.
- **gid**—is used to store the **IPC_SET** value for the effective group identification.
- **mode**—is used to store the **IPC_SET** value for the operation permissions.
- **rtrn**—is used to store the return integer from the system call which depends upon the control command or a -1 when unsuccessful.
- **semid**—is used to store and pass the semaphore set identifier to the system call.
- **semnum**—is used to store and pass the semaphore number to the system call.

Semaphores

- **cmd**—is used to store the code for the desired control command so that subsequent processing can be performed on it.
- **choice**—is used to determine which member (**uid**, **gid**, **mode**) for the **IPC_SET** control command is to be changed.
- **arg.val**—is used to pass the system call a value to set (**SETVAL**) or to store (**GETVAL**) a value returned from the system call for a single semaphore (union member).
- **arg.buf**—is a pointer passed to the system call which locates the data structure in the user memory area where the **IPC_STAT** control command is to place its return values, or where the **IPC_SET** command gets the values to set (union member).
- **arg.array**—is used to store the set of semaphore values when getting (**GETALL**) or initializing (**SETALL**) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the **sem.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

15 The **arg** union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (**SEMMSL**), a system tunable parameter.

The next important program aspect to observe is that, although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later.

If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all of the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl(S)** system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49 and 50). When it is entered, it is stored at the address of the **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56 and 57). When it is entered, it is stored at the address of the **semnum** variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59 and 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the **semnum** variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83-86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an **IPC_STAT** control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the num-

Semaphores

ber of semaphores in the set (line 91). Next, the system call is made and, upon success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for **GETVAL** above.

If the **SETALL** control command is selected (code 7), the program first performs an **IPC_STAT** control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for **GETVAL** above.

15 If the **IPC_STAT** control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191 and 192).

If the **IPC_SET** control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl(S)** system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for **GETVAL** above.

If the **IPC_RMID** control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the UNIX Operating System. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl(S)** system call follows. It is suggested that the source program file be named **semctl.c** and that the executable file be named **semctl**.

Figure 15-10 semctl() System Call Example (Sheet 1 of 5)

```

1  /*This is a program to illustrate
2  **the semaphore control, semctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct semid_ds semid_ds;
15     int c, i, length;
16     int uid, gid, mode;
17     int retm, semid, semnum, cmd, choice;
18     union semun {
19         int val;
20         struct semid_ds *buf;
21         ushort array[25];
22     } arg;

23     /*Initialize the data structure pointer.*/
24     arg.buf = &semid_ds;

25     /*Enter the semaphore ID.*/
26     printf("Enter the semid = ");
27     scanf("%d", &semid);

28     /*Choose the desired command.*/
29     printf("\nEnter the number for\n");
30     printf("the desired cmd:\n");
31     printf("GETVAL      = 1\n");
32     printf("SETVAL      = 2\n");
33     printf("GETPID      = 3\n");
34     printf("GETINQNT    = 4\n");
35     printf("GETZCNT     = 5\n");
36     printf("GETALL      = 6\n");
37     printf("SETALL      = 7\n");
38     printf("IPC_STAT    = 8\n");
39     printf("IPC_SET     = 9\n");
40     printf("IPC_RMID    = 10\n");
41     printf("Entry      = ");
42     scanf("%d", &cmd);

```

Figure 15-10 semctl() System Call Example (Sheet 2 of 5)

```

43      /*Check entries.*/
44      printf ("Insemid =%d, cmd = %d\n\n",
45              semid, cmd);

46      /*Set the command and do the call.*/
47      switch (cmd)
48      {

49      case 1: /*Get a specified value.*/
50              printf("\nEnter the semnum = ");
51              scanf("%d", &semnum);
52              /*Do the system call.*/
53              retm = semctl(semid, semnum, GETVAL, 0);
54              printf("\nThe semval = %d\n", retm);
55              break;

56      case 2: /*Set a specified value.*/
57              printf("\nEnter the semnum = ");
58              scanf("%d", &semnum);
59              printf("\nEnter the value = ");
60              scanf("%d", &arg.val);
61              /*Do the system call.*/
62              retm = semctl(semid, semnum, SETVAL, arg.val);
63              break;

64      case 3: /*Get the process ID.*/
65              retm = semctl(semid, 0, GETPID, 0);
66              printf("\nThe sempid = %d\n", retm);
67              break;

68      case 4: /*Get the number of processes
69              waiting for the semaphore to
70              become greater than its current
71              value.*/
72              printf("\nEnter the semnum = ");
73              scanf("%d", &semnum);
74              /*Do the system call.*/
75              retm = semctl(semid, semnum, GETNONT, 0);
76              printf("\nThe semncnt = %d", retm);
77              break;

78      case 5: /*Get the number of processes
79              waiting for the semaphore
80              value to become zero.*/
81              printf("\nEnter the semnum = ");
82              scanf("%d", &semnum);
83              /*Do the system call.*/
84              retm = semctl(semid, semnum, GETZCNT, 0);
85              printf("\nThe semzcnt = %d", retm);
86              break;

```


Figure 15-10 semctl() System Call Example (Sheet 3 of 5)

```

87     case 6: /*Get all of the semaphores.*/
88         /*Get the number of semaphores in
89            the semaphore set.*/
90         retm = semctl(semid, 0, IPC_STAT, arg.buf);
91         length = arg.buf->sem_nsems;
92         if(retm == -1)
93             goto ERROR;
94         /*Get and print all semaphores in the
95            specified set.*/
96         retm = semctl(semid, 0, GETALL, arg.array);
97         for (i = 0; i < length; i++)
98             {
99                 printf("%d", arg.array[i]);
100                /*Separate each
101                   semaphore.*/
102                printf("%c", ' ');
103            }
104         break;
105     case 7: /*Set all semaphores in the set.*/
106         /*Get the number of semaphores in
107            the set.*/
108         retm = semctl(semid, 0, IPC_STAT, arg.buf);
109         length = arg.buf->sem_nsems;
110         printf("Length = %d\n", length);
111         if(retm == -1)
112             goto ERROR;
113         /*Set the semaphore set values.*/
114         printf("\nEnter each value:\n");
115         for(i = 0; i < length ; i++)
116             {
117                 scanf("%d", &c);
118                 arg.array[i] = c;
119             }
120         /*Do the system call.*/
121         retm = semctl(semid, 0, SETALL, arg.array);
122         break;

123     case 8: /*Get the status for the semaphore set.*/
124         /*Get and print the current status values.*/
125         retm = semctl(semid, 0, IPC_STAT, arg.buf);
126         printf ("\nThe USER ID = %d\n",
127                arg.buf->sem_perm.uid);
128         printf ("The GROUP ID = %d\n",
129                arg.buf->sem_perm.gid);
130         printf ("The operation permissions = %o\n",
131                arg.buf->sem_perm.mode);
132         printf ("The number of semaphores in set = %d\n",
133                arg.buf->sem_nsems);
134         printf ("The last semop time = %d\n",
135                arg.buf->sem_otime);
136         printf ("The last change time = %d\n",
137                arg.buf->sem_ctime);
138         break;

```

Figure 15-10 semctl() System Call Example (Sheet 4 of 5)

```

141      case 9:      /*Select and change the desired
142                  member of the data structure.*/
143                  /*Get the current status values.*/
144                  retm = semctl(semid, 0, IPC_STAT, arg.buf);
145                  if(retm == -1)
146                      goto ERROR;
147                  /*Select the member to change.*/
148                  printf("\nEnter the number for the\n");
149                  printf("member to be changed:\n");
150                  printf("sem_perm.uid   = 1\n");
151                  printf("sem_perm.gid   = 2\n");
152                  printf("sem_perm.mode = 3\n");
153                  printf("Entry       = ");
154                  scanf("%d", &choice);
155                  switch(choice) {

156      case 1: /*Change the user ID.*/
157                  printf("\nEnter USER ID = ");
158                  scanf ("%d", &uid);
159                  arg.buf->sem_perm.uid = uid;
160                  printf("\nUSER ID = %d\n",
161                          arg.buf->sem_perm.uid);
162                  break;

163      case 2: /*Change the group ID.*/
164                  printf("\nEnter GROUP ID = ");
165                  scanf ("%d", &gid);
166                  arg.buf->sem_perm.gid = gid;
167                  printf("\nGROUP ID = %d\n",
168                          arg.buf->sem_perm.gid);
169                  break;

170      case 3: /*Change the mode portion of
171              the operation
172              permissions.*/
173                  printf("\nEnter MODE = ");
174                  scanf ("%o", &mode);
175                  arg.buf->sem_perm.mode = mode;
176                  printf("\nMODE = %o\n",
177                          arg.buf->sem_perm.mode);
178                  break;
179      }
180      /*Do the change.*/
181      retm = semctl(semid, 0, IPC_SET, arg.buf);
182      break;

```

Figure 15-10 semctl() System Call Example (Sheet 5 of 5)

```

183         case 10:    /*Remove the semid along with its
184                     data structure.*/
185                     retm = semctl(semid, 0, IPC_RMID, 0);
186                 }
187                 /*Perform the following if the call is unsuccessful.*/
188                 if(retm == -1)
189                 {
190                     ERROR:
191                     printf ("\n\nThe semctl system call failed!\n");
192                     printf ("The error number = %d\n", errno);
193                     exit (0);
194                 }
195                 printf ("\n\nThe semctl system call was successful\n");
196                 printf ("for semid = %d\n", semid);
197                 exit (0);
198             }

```

Operations on Semaphores

This section contains a detailed description of using the **semop(S)** system call along with an example program which allows all of its capabilities to be exercised.

15

Using semop

The synopsis found in the **semop(S)** entry in the *Programmer's Reference* is as follows:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;

```

The **semop(S)** system call requires three arguments to be passed to it, and it returns an integer value. Upon successful completion, a zero value is returned. When unsuccessful, a -1 is returned.

The **semid** argument must be a valid, non-negative, integer value; that is, it must have already been created by using the **semget(S)** system call.

Semaphores

The **sops** argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed
- the control command (flags)

The ****sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. **Sembuf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The **nsops** argument specifies the length of the array (the number of structures in the array). The maximum **size** of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop(S)** system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

15

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- **IPC_NOWAIT**—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which **IPC_NOWAIT** is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- **SEM_UNDO**—this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the **IPC_NOWAIT** flag set. That is, the blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations

with the `SEM_UNDO` flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (Figure 15-11) is a menu-driven program which allows all possible combinations of using the `semop(S)` system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the `shmop(S)` entry in the *Programmer's Reference*. Note that in this program `errno` is declared as an external variable, and therefore, the `errno.h` header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purpose are as follows:

- `sembuf[10]`—is used as an array buffer (line 14) to contain a maximum of ten `sembuf` type structures; ten equals `SEMOPM`, the maximum number of operations on a semaphore set for each `semop(S)` system call.
- `*sops`—is used as a pointer (line 14) to `sembuf[10]` for the system call and for accessing the structure members within the array.
- `rtrn`—is used to store the return values from the system call.
- `flags`—is used to store the code of the `IPC_NOWAIT` or `SEM_UNDO` flags for the `semop(S)` system call (line 60).
- `i`—is used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79).

Semaphores

- **nsops**—is used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM.
- **semid**—is used to store the desired semaphore set identifier for the system call.

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). **Semid** is stored at the address of the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

15

After the array is initialized, all of its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86 and 87). **Sembuf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl()** GETALL control command.

The example program for the **semop(S)** system call follows. It is suggested that the source program file be named **semop.c** and that the executable file be named **semop**.

Figure 15-11 semop(S) System Call Example (Sheet 1 of 3)

```

1  /*This is a program to illustrate
2  **the semaphore operations, semop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     struct sembuf sembuf[10], *sops;
15     char string[];
16     int retm, flags, sem_num, i, semid;
17     unsigned nsops;
18     sops = sembuf; /*Pointer to array sembuf.*/

19     /*Enter the semaphore ID.*/
20     printf("\nEnter the semid of\n");
21     printf("the semaphore set to\n");
22     printf("be operated on = ");
23     scanf("%d", &semid);
24     printf("\nsemid = %d", semid);

25     /*Enter the number of operations.*/
26     printf("\nEnter the number of semaphore\n");
27     printf("operations for this set = ");
28     scanf("%d", &nsops);
29     printf("\nnsops = %d", nsops);

30     /*Initialize the array for the
31     number of operations to be performed.*/
32     for(i = 0; i < nsops; i++, sops++)
33     {

34         /*This determines the semaphore in
35         the semaphore set.*/
36         printf("\nEnter the semaphore\n");
37         printf("number (sem_num) = ");
38         scanf("%d", &sem_num);
39         sops->sem_num = sem_num;
40         printf("\nThe sem_num = %d", sops->sem_num);

```

Figure 15-11 semop(S) System Call Example (Sheet 2 of 3)

```

41      /*Enter a (-)number to decrement,
42      an unsigned number (no +) to increment,
43      or zero to test for zero. These values
44      are entered into a string and converted
45      to integer values.*/
46      printf("\nEnter the operation for\n");
47      printf("the semaphore (sem_op) = ");
48      scanf("%s", string);
49      sops->sem_op = atoi(string);
50      printf("\nsem_op = %d\n", sops->sem_op);

51      /*Specify the desired flags.*/
52      printf("\nEnter the corresponding\n");
53      printf("number for the desired\n");
54      printf("flags:\n");
55      printf("No flags                = 0\n");
56      printf("IPC_NOWAIT                    = 1\n");
57      printf("SEM_UNDO                      = 2\n");
58      printf("IPC_NOWAIT and SEM_UNDO      = 3\n");
59      printf("Flags                        = ");
60      scanf("%d", &flags);

61      switch(flags)
62      {
63      case 0:
64          sops->sem_flg = 0;
65          break;
66      case 1:
67          sops->sem_flg = IPC_NOWAIT;
68          break;
69      case 2:
70          sops->sem_flg = SEM_UNDO;
71          break;
72      case 3:
73          sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74          break;
75      }
76      printf("\nFlags = 0\n", sops->sem_flg);
77  }

78      /*Print out each structure in the array.*/
79      for(i = 0; i < nsops; i++)
80      {
81          printf("\nsem_num = %d\n", sembuf[i].sem_num);
82          printf("sem_op = %d\n", sembuf[i].sem_op);
83          printf("sem_flg = %d\n", sembuf[i].sem_flg);
84          printf("%c", ' ');
85      }

```

Figure 15-11 semop(S) System Call Example (Sheet 3 of 3)

```
86      sops = sembuf; /*Reset the pointer to
87                      sembuf[0].*/

88      /*Do the semop system call.*/
89      retm = semop(semid, sops, nsops);
90      if(retm == -1) {
91          printf("\nSemop failed. ");
92          printf("Error = %d\n", errno);
93      }
94      else {
95          printf ("\nSemop was successful\n");
96          printf("for semid = %d\n", semid);

97          printf("Value returned = %d\n", retm);
98      }
99  }
```

Shared Memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and, consequently, the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget(S)** system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat(S)** — shared memory attach
- **shmdt(S)** — shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl(S)** system call. However, the creating process remains the creator until the facility is removed, or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl(S)** system call.

System calls, which are documented in the *Programmer's Reference*, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX Operating System at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time
- last detach time
- last change time

Shared Memory

The C Programming Language data structure definition for the shared memory segment data structure is located in the `/usr/include/sys/shm.h` header file. It is as follows:

```
/*
**   There is a shared mem id data structure for
**   each segment in the system.
*/

struct shmid_ds {
    struct ipc_perm    shm_perm;    /* operation permission struct */
    int                shm_segsz;    /* segment size */
    struct region      *shm_reg;    /* ptr to region structure */
    char               pad[4];      /* for swap compatibility */
    ushort             shm_lpid;    /* pid of last shmop */
    ushort             shm_cpid;    /* pid of creator */
    ushort             shm_nattch;  /* used only for shminfo */
    ushort             shm_cnattch; /* used only for shminfo */
    time_t             shm_atime;   /* last shmat time */
    time_t             shm_dtime;   /* last shmdt time */
    time_t             shm_ctime;   /* last change time */
};
```

Note that the `shm_perm` member of this structure uses `ipc_perm` as a template. The breakout for the operation permissions data structure is shown in Figure 15-1.

The `ipc_perm` data structure is the same for all IPC facilities, and it is located in the `#include <sys/ipc.h>` header file. It is shown in the introduction section of "Messages."

Figure 15-12 is a table that shows the shared memory state information.

Figure 15-12 Shared Memory State Information

Shared Memory States			
Lock Bit	Swap Bit	Allocated Bit	Implied State
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Locked Incore
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

The implied states of Figure 15-12 are as follows:

- **Unallocated Segment**—the segment associated with this segment descriptor has not been allocated for use.
- **Incore**—the shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.
- **On Disk**—the shared segment associated with this segment descriptor is currently resident on the swap device.
- **Locked Incore**—the shared segment associated with this segment descriptor is currently locked in memory and will not be a candidate for swapping until the segment is unlocked. Only the super-user may lock and unlock a shared segment.
- **Unused**—this state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget(S)** system call is used to perform two tasks when only the **IPC_CREAT** flag is set in the **shmflg** argument that it receives:

- to get a new **shmid** and create an associated shared memory segment data structure for it
- to return an existing **shmid** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget(S)** system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it, provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (**IPC_PRIVATE** = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it, unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the **KEY** field for the **shmid** is all zeros.

Shared Memory

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (**IPC_EXCL**) can be specified (**set**) in the **shmflg** argument passed to the system call. The details for using this system call are discussed in the "Using **shmget**" section of this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Shared Memory" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop()**] and control [**shmctl(S)**] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat(S)** and **shmdt(S)**. Refer to the "Operations for Shared Memory" section in this chapter for details of these system calls.

Shared memory segment control is done by using the **shmctl(S)** system call. It permits you to control the shared memory facility in the following ways:

- by determining the associated data structure status for a shared memory segment (**shmid**)
- by changing operation permissions for a shared memory segment
- by removing a particular **shmid** from the UNIX Operating System along with its associated shared memory segment data structure
- by locking a shared memory segment in memory
- by unlocking a shared memory segment.

Refer to the "Controlling Shared Memory" section in this chapter for details of the **shmctl(S)** system call.

Getting Shared Memory Segments

This section gives a detailed description of using the **shmget(S)** system call along with an example program illustrating its use.

Using shmget

The synopsis found in the **shmget(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

All of these include files are located in the **/usr/include/sys** directory of the UNIX Operating System. The following line in the synopsis informs you that **shmget(S)** is a function with three formal arguments that returns an integer type value, upon successful completion (**shmid**).

```
int shmget (key, size, shmflg)
```

The next two lines declare the types of the formal arguments. The variable **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

```
key_t key;
int size, shmflg;
```

The integer returned from this function upon successful completion is the shared memory identifier (**shmid**) that was discussed earlier.

As declared, the process calling the **shmget(S)** system call must supply three arguments to be passed to the formal **key**, **size**, and **shmflg** arguments.

Shared Memory

A new **shmid** with an associated shared memory data structure is provided if one of the following conditions exists:

- **key** is equal to **IPC_PRIVATE**
- **key** is passed a unique hexadecimal integer, and **shmflg** ANDed with **IPC_CREAT** is **TRUE**.

The value passed to the **shmflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes, and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as "operation permissions." Figure 15-13 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

15

Figure 15-13 Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **shm.h** header file which can be used for the user (OWNER). They are as follows:

```
SHM_R 0400
SHM_W 0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 15-14 contains the names of the constants that apply to the **shmget()** system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

Figure 15-14 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for the **shmflg** argument is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; bit positions and values for the control commands in relation to those of the operation permissions make this possible. An example of determining the **shmflg** argument follows.

	Octal Value	Binary Value
IPC_CREAT	0 1 0 0 0	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
Read by User	0 0 4 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
shmflg	0 1 4 0 0	0 0 0 0 0 0 1 1 0 0 0 0 0 0 0

The **shmflg** value can easily be set by using the names of the flags in conjunction with the octal operation permissions value:

```
shmld = shmget (key, size, (IPC_CREAT | 0400));
shmld = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **shmget(S)** entry in the *Programmer's Reference*, success or failure of this system call depends upon the argument values for **key**, **size**, and **shmflg** or system tunable parameters. The system call will attempt to return a new **shmld** if one of the following conditions exists:

- **key** is equal to **IPC_PRIVATE (0)**.
- **key** does not already have a **shmld** associated with it, and (**shmflg** & **IPC_CREAT**) is TRUE (not zero).

The **key** argument can be set to **IPC_PRIVATE** in the following ways:

```
shmld = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmld = shmget ( 0 , size, shmflg);
```


Shared Memory

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system tunable parameter always causes a failure. The SHMMNI system tunable parameter determines the maximum number of unique shared memory segments (**shmid**s) in the UNIX Operating System.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and **IPC_CREAT** is TRUE (not zero). This means that the **key** is unique (not in use) within the UNIX Operating System for this facility type and that the **IPC_CREAT** flag is set (**shmflg** | **IPC_CREAT**). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
shmflg = x 1 x x x    (x = immaterial)
& IPC_CREAT = 0 1 0 0 0
result = 0 1 0 0 0    (not zero)
```

Because the result is not zero, the flag is set or TRUE. SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** if the **key** equals zero (IPC_PRIVATE).

The system call will fail if the value for the **size** argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment sizes.

Refer to the **shmget(S)** manual page for specific, associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 15-15) is a menu-driven program which allows all possible combinations of using the **shmget(S)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required header files as specified by the **shmget(S)** entry in the *Programmer's Reference*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purposes are as follows:

- **key**—is used to pass the value for the desired **key**.
- **opperm**—is used to store the desired operation permissions.
- **flags**—is used to store the desired control commands (flags).
- **opperm_flags**—is used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument.
- **shmid**—is used for returning the message queue identification number for a successful system call or the error code (-1) for an unsuccessful one.
- **size**—is used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made next, and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 58). If **shmid** equals -1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60 and 61).

Shared Memory

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget(S)** system call follows. It is suggested that the source program file be named **shmget.c** and that the executable file be named **shmget**.

Figure 15-15 shmget(S) System Call Example (Sheet 1 of 2)

```
1  /*This is a program to illustrate
2  **the shared memory get, shmget(),
3  **system call capabilities.*/

4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <errno.h>

8  /*Start of main C language program*/
9  main()
10 {
11     key_t key;          /*declare as long integer*/
12     int opperm, flags;
13     int shmid, size, opperm_flags;
14     /*Enter the desired key*/
15     printf("Enter the desired key in hex = ");
16     scanf("%x", &key);

17     /*Enter the desired octal operation
18     permissions.*/
19     printf("\nEnter the operation\n");
20     printf("permissions in octal = ");
21     scanf("%o", &opperm);

22     /*Set the desired flags.*/
23     printf("\nEnter corresponding number to\n");
24     printf("set the desired flags:\n");
25     printf("No flags                = 0\n");
26     printf("IPC_CREAT                    = 1\n");
27     printf("IPC_EXCL                      = 2\n");
28     printf("IPC_CREAT and IPC_EXCL        = 3\n");
29     printf("Flags                          = ");
30     /*Get the flag(s) to be set.*/
31     scanf("%d", &flags);

32     /*Check the values.*/
33     printf("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34           key, opperm, flags);
```

Figure 15-15 shmget(S) System Call Example (Sheet 2 of 2)

```

35      /*Incorporate the control fields (flags) with
36      the operation permissions*/
37      switch (flags)
38      {
39      case 0: /*No flags are to be set.*/
40          opperm_flags = (opperm | 0);
41          break;
42      case 1: /*Set the IPC_CREAT flag.*/
43          opperm_flags = (opperm | IPC_CREAT);
44          break;
45      case 2: /*Set the IPC_EXCL flag.*/
46          opperm_flags = (opperm | IPC_EXCL);
47          break;
48      case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
49          opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50      }

51      /*Get the size of the segment in bytes.*/
52      printf ("\nEnter the segment");
53      printf ("\nsize in bytes = ");
54      scanf ("%d", &size);

55      /*Call the shmget system call.*/
56      shmid = shmget (key, size, opperm_flags);

57      /*Perform the following if the call is unsuccessful.*/
58      if(shmid == -1)
59      {
60          printf ("\nThe shmget system call failed!\n");
61          printf ("The error number = %d\n", errno);
62      }
63      /*Return the shmid upon successful completion.*/
64      else
65          printf ("\nThe shmid = %d\n", shmid);
66      exit(0);
67  }

```

Controlling Shared Memory

This section gives a detailed description of using the **shmctl(S)** system call along with an example program which allows all of its capabilities to be exercised.

Shared Memory

Using shmctl

The synopsis found in the **shmctl(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmids, cmd, buf)
int shmids, cmd;
struct shmids *buf;
```

The **shmctl(S)** system call requires three arguments to be passed to it, and it returns an integer value. Upon successful completion, a zero value is returned. When unsuccessful, a -1 is returned.

The **shmids** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(S)** system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

- **IPC_STAT**—returns the status information contained in the associated data structure for the specified **shmids** and places it in the data structure pointed to by the ***buf** pointer in the user memory area
- **IPC_SET**—for the specified **shmids**, sets the effective user and group identification, and operation permissions
- **IPC_RMID**—removes the specified **shmids** along with its associated shared memory segment data structure
- **SHM_LOCK**—locks the specified shared memory segment in memory; must be super-user
- **SHM_UNLOCK**—unlocks the shared memory segment from memory; must be super-user.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an **IPC_SET** or **IPC_RMID** control command. Only the super-user can perform a **SHM_LOCK** or **SHM_UNLOCK** control command. A process must have read permission to perform the **IPC_STAT** control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this

program, read the "Using **shmget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-16) is a menu-driven program which allows all possible combinations of using the **shmctl(S)** system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmctl(S)** entry in the *Programmer's Reference*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal, since they are local to the program. Variables declared for this program and their purposes are as follows:

- **uid**—is used to store the IPC_SET value for the effective user identification.
- **gid**—is used to store the IPC_SET value for the effective group identification.
- **mode**—is used to store the IPC_SET value for the operation permissions.
- **rtrn**—is used to store the return integer value from the system call.
- **shmid**—is used to store and pass the shared memory segment identifier to the system call.
- **command**—is used to store the code for the desired control command so that subsequent processing can be performed on it.
- **choice**—is used to determine which member for the IPC_SET control command is to be changed.

Shared Memory

- **shm_id_ds**—is used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed.
- ***buf**—is a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set.

Note that the **shm_id_ds** data structure in this program (line 16) uses the data structure located in the **shm.h** header file of the same name as a template for its declaration. This is an example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shm_id_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

15 First, the program prompts for a valid shared memory segment identifier which is stored at the address of the **shm_id** variable (lines 18-20). This is required for every **shmctl(S)** system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39 and 40), and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148 and 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing to do is get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice

variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending upon success or failure, the program returns the same messages as for `IPC_STAT` above.

If the `IPC_RMID` control command (code 3) is selected, the system call is performed (lines 132-135), and the `shmid` along with its associated message queue and data structure are removed from the UNIX Operating System. Note that the `*buf` pointer is not required as an argument to perform this control command and its value can be zero or `NULL`. Depending upon success or failure, the program returns the same messages as for the other control commands.

If the `SHM_LOCK` control command (code 4) is selected, the system call is performed (lines 137 and 138). Depending upon success or failure, the program returns the same messages as for the other control commands.

If the `SHM_UNLOCK` control command (code 5) is selected, the system call is performed (lines 140-142). Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the `shmctl(S)` system call follows. It is suggested that the source program file be named `shmctl.c` and that the executable file be named `shmctl`.

When compiling C programs that use floating point operations, the `-f` option should be used on the `cc` command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail. The `-f` option is not required, however, on your computer.

Shared Memory

Figure 15-16 shmctl(S) System Call Example (Sheet 1 of 4)

```
1  /*This is a program to illustrate
2  **the shared memory control, shmctl(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>

10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int uid, gid, mode;
15     int rtm, shmid, command, choice;
16     struct shm_id_s shm_id_s, *buf;
17     buf = &shm_id_s;

18     /*Get the shmid, and command.*/
19     printf("Enter the shmid = ");
20     scanf("%d", &shmid);
21     printf("\nEnter the number for\n");
22     printf("the desired command:\n");

23     printf("IPC_STAT    = 1\n");
24     printf("IPC_SET     = 2\n");
25     printf("IPC_RMID    = 3\n");
26     printf("SHM_LOCK    = 4\n");
27     printf("SHM_UNLOCK  = 5\n");
28     printf("Entry      = ");
29     scanf("%d", &command);
```

Figure 15-16 shmctl(S) System Call Example (Sheet 2 of 4)

```

30      /*Check the values.*/
31      printf ("\nshmid =%d, command = %d\n",
32             shmid, command);

33      switch (command)
34      {
35      case 1: /*Use shmctl() to duplicate
36              the data structure for
37              shmid in the shmid_ds area pointed
38              to by buf and then print it out.*/
39          rtm = shmctl(shmid, IPC_STAT,
40                     buf);
41          printf ("\nThe USER ID = %d\n",
42                 buf->shm_perm.uid);
43          printf ("The GROUP ID = %d\n",
44                 buf->shm_perm.gid);
45          printf ("The creator's ID = %d\n",
46                 buf->shm_perm.cuid);
47          printf ("The creator's group ID = %d\n",
48                 buf->shm_perm.cgid);
49          printf ("The operation permissions = %o\n",
50                 buf->shm_perm.mode);
51          printf ("The slot usage sequence\n");
52          printf ("number = %x\n",
53                 buf->shm_perm.seq);
54          printf ("The key= %x\n",
55                 buf->shm_perm.key);
56          printf ("The segment size = %d\n",
57                 buf->shm_segsz);
58          printf ("The pid of last shmop = %d\n",
59                 buf->shm_lpid);
60          printf ("The pid of creator = %d\n",
61                 buf->shm_cpid);
62          printf ("The current # attached = %d\n",
63                 buf->shm_nattach);
64          printf ("The in memory # attached = %d\n",
65                 buf->shm_cnattach);
66          printf ("The last shmat time = %d\n",
67                 buf->shm_atime);
68          printf ("The last shmdt time = %d\n",
69                 buf->shm_dtime);
70          printf ("The last change time = %d\n",
71                 buf->shm_ctime);
72          break;

          /* Lines 73 - 87 deleted */

```


Shared Memory

Figure 15-16 shmctl(S) System Call Example (Sheet 3 of 4)

```
88      case 2:    /*Select and change the desired
89                  member(s) of the data structure.*/

90      /*Get the original data for this shmid
91          data structure first.*/
92      rtm = shmctl(shmid, IPC_STAT, buf);

93      printf("\nEnter the number for the\n");
94      printf("member to be changed:\n");
95      printf("shm_perm.uid   = 1\n");
96      printf("shm_perm.gid   = 2\n");
97      printf("shm_perm.mode  = 3\n");
98      printf("Entry       = ");
99      scanf("%d", &choice);
100     /*Only one choice is allowed per
101         pass as an illegal entry will
102         cause repetitive failures until
103         shmid_ds is updated with
104         IPC_STAT.*/
105     switch(choice){
106     case 1:
107         printf("\nEnter USER ID = ");
108         scanf ("%d", &uid);
109         buf->shm_perm.uid = uid;
110         printf("\nUSER ID = %d\n",
111             buf->shm_perm.uid);
112         break;

113     case 2:
114         printf("\nEnter GROUP ID = ");
115         scanf ("%d", &gid);
116         buf->shm_perm.gid = gid;
117         printf("\nGROUP ID = %d\n",
118             buf->shm_perm.gid);
119         break;

120     case 3:
121         printf("\nEnter MODE = ");
122         scanf ("%o", &mode);
123         buf->shm_perm.mode = mode;
124         printf("\nMODE = 0%o\n",
125             buf->shm_perm.mode);
126         break;
127     }
128     /*Do the change.*/
129     rtm = shmctl(shmid, IPC_SET,
130         buf);
131     break;
```

Figure 15-16 shmctl(S) System Call Example (Sheet 4 of 4)

```

132     case 3:    /*Remove the shmid along with its
133                associated
134                data structure.*/
135         rtm = shmctl(shmid, IPC_RMID, NULL);
136         break;

137     case 4: /*Lock the shared memory segment*/
138         rtm = shmctl(shmid, SHM_LOCK, NULL);
139         break;
140     case 5: /*Unlock the shared memory
141            segment.*/
142         rtm = shmctl(shmid, SHM_UNLOCK, NULL);
143         break;
144     }
145     /*Perform the following if the call is unsuccessful.*/
146     if(rtm == -1)
147     {
148         printf ("\nThe shmctl system call failed!\n");
149         printf ("The error number = %d\n", errno);
150     }
151     /*Return the shmid upon successful completion.*/
152     else
153         printf ("\nShmctl was successful for shmid = %d\n",
154             shmid);
155     exit (0);
156 }

```

Operations for Shared Memory

This section gives a detailed description of using the **shmat(S)** and **shmdt(S)** system calls. It also provides an example program which allows all of their capabilities to be exercised.

Shared Memory

Using shmop

The synopsis found in the **shmop(S)** entry in the *Programmer's Reference* is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat(S)** system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. Upon successful completion, this value will be the address in core memory where the process is attached to the shared memory segment. When unsuccessful the value will be a -1.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget(S)** system call.

The **shmaddr** argument can be zero or user-supplied when passed to the **shmat(S)** system call. If it is zero, the UNIX Operating System picks the address of where the shared memory segment will be attached. If it is user-supplied, the address must be a valid address that the UNIX Operating System would pick. The following table illustrates some typical address ranges for your computer:

80286	80386
0x01F70000	0x80400000
0x02070000	0x80800000
0x020F0000	0x80C00000
0x02170000	0x81000000

Note that these addresses are in chunks of 0x80000 hexadecimal (for the 80286 Computer) and 0x1000 hexadecimal (for the 80386 Computer). It would be wise to let the operating system pick addresses so as to improve portability.

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat()** system call.

Further details are discussed in the example program for **shmop()**. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt(S)** system call requires one argument to be passed to it, and it returns an integer value. Upon successful completion, zero is returned. When unsuccessful, a -1 is returned.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 15-17) is a menu-driven program which allows all possible combinations of using the **shmat(S)** and **shmdt(S)** system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop(S)** entry in the *Programmer's Reference*. Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and are perfectly legal since they are local to the program. Variables declared for this program and their purposes are as follows:

- **flags**—is used to store the codes of SHM_RND or SHM_RDONLY for the **shmat(S)** system call.

Shared Memory

- **addr**—is used to store the address of the shared memory segment for the **shmat(S)** and **shmdt(S)** system calls.
- **i**—is used as a loop counter for attaching and detaching.
- **attach**—is used to store the desired number of attach operations.
- **shmid**—is used to store and pass the desired shared memory segment identifier.
- **shmflg**—is used to pass the value of flags to the **shmat(S)** system call.
- **retrn**—is used to store the return values from both system calls.
- **detach**—is used to store the desired number of detach operations.

This example program combines both the **shmat(S)** and **shmdt(S)** system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

15

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the **attach** variable (lines 17-21).

A loop is entered using the **attach** variable and the **i** counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat(S)** system call (lines 46-57). The system call is made (line 60). If successful, a message so stating is displayed along with the attach address (lines 66-68). If unsuccessful, a message so stating is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the *i* counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the **addr** variable (line 84). Then, the **shmdt(S)** system call is performed (line 87). If successful, a message so stating is displayed along with the address that the segment was detached from (lines 92 and 93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the **shmop(S)** system calls follows. It is suggested that the program be put into a source file called **shmop.c** and then into an executable file called **shmop**.

Figure 15-17 shmop() System Call Example (Sheet 1 of 3)

```
1  /*This is a program to illustrate
2  **the shared memory operations, shmop(),
3  **system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
10 /*Start of main C language program*/
11 main()
12 {
13     extern int errno;
14     int flags, addr, i, attach;
15     int shmid, shmflg, retm, detach;

16     /*Loop for attachments by this process.*/
17     printf("Enter the number of\n");
18     printf("attachments for this\n");
19     printf("process (1-4).\n");
20     printf("      Attachments = ");

21     scanf("%d", &attach);
22     printf("Number of attaches = %d\n", attach);
```

Figure 15-17 shmop() System Call Example (Sheet 2 of 3)

```

23     for(i = 1; i <= attach; i++) {
24         /*Enter the shared memory ID.*/
25         printf("\nEnter the shmid of\n");
26         printf("the shared memory segment to\n");
27         printf("be operated on = ");
28         scanf("%d", &shmid);
29         printf("\nshmid = %d\n", shmid);

30         /*Enter the value for shmaddr.*/
31         printf("\nEnter the value for\n");
32         printf("the shared memory address\n");
33         printf("in hexadecimal:\n");
34         printf("          Shmaddr = ");
35         scanf("%x", &addr);
36         printf("The desired address = 0x%x\n", addr);

37         /*Specify the desired flags.*/
38         printf("\nEnter the corresponding\n");
39         printf("number for the desired\n");
40         printf("flags:\n");
41         printf("SHM_RND                = 1\n");
42         printf("SHM_RDONLY                = 2\n");
43         printf("SHM_RND and SHM_RDONLY = 3\n");
44         printf("          Flags          = ");
45         scanf("%d", &flags);

46         switch(flags)
47         {
48             case 1:
49                 shmflg = SHM_RND;
50                 break;
51             case 2:
52                 shmflg = SHM_RDONLY;
53                 break;
54             case 3:
55                 shmflg = SHM_RND | SHM_RDONLY;
56                 break;
57         }
58         printf("\nFlags = 0x%x\n", shmflg);

```

Figure 15-17 shmop() System Call Example (Sheet 3 of 3)

```

59      /*Do the shmat system call.*/
60      retm = (int)shmat(shmid, addr, shmflg);
61      if(retm == -1) {
62          printf("\nShmat failed. ");
63          printf("Error = %d\n", errno);
64      }
65      else {
66          printf ("\nShmat was successful\n");
67          printf("for shmid = %d\n", shmid);
68          printf("The address = 0x%x\n", retm);
69      }
70  }

71      /*Loop for detachments by this process.*/
72      printf("Enter the number of\n");
73      printf("detachments for this\n");
74      printf("process (1-4).\n");
75      printf("      Detachments = ");

76      scanf("%d", &detach);
77      printf("Number of attaches = %d\n", detach);
78      for(i = 1; i <= detach; i++) {

79          /*Enter the value for shmaddr.*/
80          printf("\nEnter the value for\n");
81          printf("the shared memory address\n");
82          printf("in hexadecimal:\n");
83          printf("      Shmaddr = ");
84          scanf("%x", &addr);
85          printf("The desired address = 0x%x\n", addr);

86          /*Do the shmdt system call.*/
87          retm = (int)shmdt(addr);
88          if(retm == -1) {
89              printf("Error = %d\n", errno);
90          }
91          else {
92              printf ("\nShmdt was successful\n");
93              printf("for address = 0x%x\n", addr);

94          }
95      }
96  }

```


Chapter 16

Common Object File Format (COFF)

The Common Object File Format (COFF)	16-1
Definitions and Conventions	16-2
File Header	16-4
Optional Header Information	16-6
Section Headers	16-8
Sections	16-10
Relocation Information	16-11
Line Numbers	16-13
Symbol Table	16-14
Symbol Names	16-19
Storage Classes	16-20
Storage Classes for Special Symbols	16-21
Symbol Value Field	16-22
Section Number Field	16-23
Section Numbers and Storage Classes	16-24
Type Entry	16-24
Type Entries and Storage Classes	16-26
Structure for Symbol Table Entries	16-28
File Names	16-30
Sections	16-30
Tag Names	16-31
End of Structures	16-31
Functions	16-32
Arrays	16-32
End of Blocks and Functions	16-33
Beginning of Blocks and Functions	16-33
Names Related to Structures, Unions, and Enumerations	16-33
Auxiliary Entry Declaration	16-35
String Table	16-36
Access Routines	16-37

The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF) used on your computer with the UNIX Operating System. COFF is the format of the output file produced by the assembler, `as`, and the link editor, `ld`.

The following are some key features of COFF:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

The Common Object File Format (COFF)

Figure 16-1 shows the overall structure.

Figure 16-1 Object File Format

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **-s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the **strip** command. The line number information does not appear unless the program is compiled with the **-g** option of the **cc** command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX Operating System loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

Note

It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but for COFF on UNIX Systems, the physical address is equivalent to the virtual address.

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer, with the intent of creating an object file that can be executed on another computer. The term, target machine, refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the exact same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Figure 16-2. The last 2 bytes are flags that are used by **ld** and object file utilities.

Figure 16-2 File Header Contents

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see Figure 16-3)

Magic Numbers

The magic number specifies the target machine on which the object file is executable.

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h** and are shown in Figure 16-3.

Figure 16-3 File Header Flags

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR16WR	0000200	16-bit byte reversed word
F_AR32WR	0000400	32-bit byte reversed word

File Header Declaration

The C structure declaration for the file header is given in Figure 16-4. This declaration may be found in the header file **filehdr.h**.

Figure 16-4 File Header Declaration

```

struct filehdr
{
    unsigned short  f_magic; /* magic number */
    unsigned short  f_nscns; /* number of section */

    long            f_timdat; /* time and date stamp */

    long            f_symptr; /* file ptr to symbol table */

    long            f_nsyms; /* number entries in the symbol table */

    unsigned short  f_opthdr; /* size of optional header */

    unsigned short  f_flags; /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
    
```

Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field `f_opthdr`.

Standard UNIX System `a.out` Header

By default, files produced by the link editor for a UNIX System always have a standard UNIX System `a.out` header in the optional header field. The UNIX System `a.out` header is 28 bytes. The fields of the optional header are described in Figure 16-5.

Figure 16-5 Optional Header Contents

Bytes	Declaration	Name	Description
0-1	short	magic	Magic number
2-3	short	vstamp	Version stamp
4-7	long int	tsize	Size of text in bytes
8-11	long int	dsize	Size of initialized data in bytes
12-15	long int	bsize	Size of uninitialized data in bytes
16-19	long int	entry	Entry point
20-23	long int	text_start	Base address of text
24-27	long int	data_start	Base address of data

Whereas the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the UNIX System V/386 Release 3.2 operating system are given in Figure 16-6.

Figure 16-6 UNIX System Magic Numbers

Value	Meaning
0407	Text segment is not write-protected or sharable; data segment is contiguous with the text segment.
0410	Data segment starts at the next segment following the text segment and the text segment is write-protected.
0413	Text and data segments are aligned within a.out so it can be directly paged.
0443	Defines a.out to be a target shared library.

Optional Header Declaration

The C language structure declaration currently used for the UNIX System **a.out** file header is given in Figure 16-7. This declaration may be found in the header file **aouthdr.h**.

Figure 16-7 **aouthdr** Declaration

```
typedef struct aouthdr
{
    short    magic;        /* magic number */
    short    vstamp;       /* version stamp */
    long     tsize;        /* text size in bytes, padded */
                                /* to full word boundary */
    long     dsize;        /* initialized data size */
    long     bsize;        /* uninitialized data size */
    long     entry;        /* entry point */
    long     text_start;    /* base of text for this file */
    long     data_start     /* base of data for this file */
} AOUTHDR;
```


Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 16-8.

Figure 16-8 Section Header Contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Figure 16-9)

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the UNIX System function `fseek(S)`.

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 16-9.

Figure 16-9 Section Header Flags

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For .lib section (treated like STYP_INFO)

Section Header Declaration

The C structure declaration for the section headers is described in Figure 16-10. This declaration may be found in the header file **scnhdr.h**.

Figure 16-10 Section Header Declaration

```
struct scnhdr
{
    char        s_name[8];        /* section name */
    long         s_paddr;          /* physical address */
    long         s_vaddr;          /* virtual address */
    long         s_size;           /* section size */
    long         s_scnptr;         /* file ptr to section raw data */

    long         s_relptr;         /* file ptr to relocation */
    long         s_lnnoptr;        /* file ptr to line number */

    unsigned short s_nreloc;       /* number of relocation entries */
    unsigned short s_nlnno;       /* number of line number entries */

    long         s_flags;          /* flags */
};

#define SCNHDR struct scnhdr
#define SCNHSZ sizeof(SCNHDR)
```

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the **STYP_NOLOAD** and **STYP_DSECT** sections.

Sections

Figure 16-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor **SECTIONS** directives (see Chapter 6) allow users to do the following, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections.

If no **SECTIONS** directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together, the output object file contains a single **.text** section made up of the combined input **.text** sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 16-11.

Figure 16-11 Relocation Section Contents

Bytes	Declaration	Name	Description
0-3	long int	r_vaddr	(Virtual) address of reference
4-7	long int	r_symndx	Symbol table index
8-9	unsigned short	r_type	Relocation type

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 16-12.

The Common Object File Format (COFF)

Figure 16-12 Relocation Types

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR16 *	01	Direct, 16-bit reference to a symbol's virtual address.
R_REL16 *	02	"PC-relative", 16-bit reference to a symbol's virtual address. Relative references occur in instructions such as jumps and calls.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_SEG12 *	011	Direct, 16-bit reference to the segment-selector bits of a 32-bit virtual address.
R_PCRLONG †	024	"PC_relative", 32-bit reference to a symbol's virtual address.
* 80286 Computer only.		
† 80386 Computer only.		

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 16-13. This declaration may be found in the header file **reloc.h**.

Figure 16-13 Relocation Entry Declaration

```
struct reloc
{
    long          r_vaddr;    /* virtual address of reference */
    long          r_symndx;    /* index into symbol table */
    unsigned short r_type;     /* relocation type */
};

#define RELOC      struct reloc
#define RELSZ      10
```

Line Numbers

When invoked with the `-g` option, the `cc` command causes an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like `sdb`. All line numbers in a section are grouped by function as shown in Figure 16-14.

Figure 16-14 Line Number Grouping

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

16

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 16-15.

Figure 16-15 Line Number Entry Declaration

```
struct lineno
{
    union
    {
        long l_symndx; /* symtbl index of func name */
        long l_paddr; /* paddr of line number */
    } l_addr;
    unsigned short l_lnno; /* line number */
};

#define LINENO struct lineno
#define LINESZ 6
```

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 16-16.

Figure 16-16 COFF Symbol Table

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
...
statics
...
filename 2
function 1
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

The word *statics* in Figure 16-16 means symbols defined with the C language storage class *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table contains some special symbols that are generated by *as* and other tools. These symbols are given in Figure 16-17.

Figure 16-17 Special Symbols in the Symbol Table

Symbol	Meaning
.file	filename
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section .text
edata	next available address after the end of the output section .data
end	next available address after the end of the output section .bss

16

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.xfake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, { and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 16-18.

Figure 16-18 Special Symbols (**.bb** and **.eb**)

<u>.bb</u>
local symbols
for that block
<u>.eb</u>

Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested (see Figure 16-19).

Figure 16-19 Nested blocks

```
{
    /* block 1 */
    int i;
    char c;
    ...
    {
        /* block 2 */
        long a;
        ...
        {
            /* block 3 */
            int x;
            ....
        }
        /* block 3 */
    }
    /* block 2 */
    {
        /* block 4 */
        long i;
        ...
    }
    /* block 4 */
}
/* block 1 */
```

16

The symbol table would look like Figure 16-20.

Figure 16-20 Example of the Symbol Table

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 16-21.

Figure 16-21 Symbols for Functions

function name
.bf
local symbol
.ef

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 16-22. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Figure 16-22 Symbol Table Entry Format

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These 8 bytes contain either a symbol name or an index to a symbol
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnm	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 16-23.

Figure 16-23 Name Field

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Special symbols generated by the C Compilation System are discussed above in "Special Symbols."

The Common Object File Format (COFF)

Storage Classes

The storage class field has one of the values described in Figure 16-24. These `#define`'s may be found in the header file `storclass.h`.

Figure 16-24 Storage Classes

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of a function
C_NULL	0	-
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARAM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

All of these storage classes except for `C_ALIAS` and `C_HIDDEN` are generated by the `cc` or `as` commands. The compress utility, `cprs`, generates the `C_ALIAS` mnemonic. This utility (described in the *Programmer's Reference*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class `C_HIDDEN` is not used by any UNIX System tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 16-25.

Figure 16-25 Storage Class by Special Symbols

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Also some storage classes are used only for certain special symbols. They are summarized in Figure 16-26.

Figure 16-26 Restricted Storage Classes

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

The Common Object File Format (COFF)

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 16-27.

Figure 16-27 Storage Class and Value

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

If a symbol has storage class `C_FILE`, the value of that symbol equals the symbol table entry index of the next `.file` symbol. That is, the `.file` entries form a one-way linked list in the symbol table. If there are no more `.file` entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 16-28.

Figure 16-28 Section Number

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and .eos symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply-defined external symbol (i.e., FORTRAN common or an uninitialized variable-defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0, and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the .bss section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

The Common Object File Format (COFF)

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 16-29.

Figure 16-29 Section Number and Storage Class

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is

d6	d5	d4	d3	d2	d1	typ
----	----	----	----	----	----	-----

Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 16-30.

Figure 16-30 Fundamental Types

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_ARG	1	Function argument (used only by compiler)
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 16-31.

Figure 16-31 Derived Types

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```


The Common Object File Format (COFF)

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Figure 16-32 shows the type entries that are legal for each storage class.

Figure 16-32 Type Entries by Storage Class

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE

(Continued on next page.)

Figure 16-33 Type Entries by Storage Class

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

The Common Object File Format (COFF)

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 16-33. This declaration may be found in the header file `syms.h`.

Figure 16-34 Symbol Table Entry Declaration

```
struct syment
{
    union
    {
        char          _n_name[SYMNMLEN];    /* symbol name*/
        struct
        {
            long       _n_zeroes;    /* symbol name */
            long       _n_offset;    /* location in string table */
        } _n_n;
        char          *_n_nptr[2]; /* allows overlaying */
    } _n;
    unsigned long     n_value;        /* value of symbol */
    short             n_scnnum;       /* section number */
    unsigned short     n_type;        /* type and derived */
    char              n_sclass;       /* storage class */
    char              n_numaux;       /* number of aux entries */
};

#define n_name          _n._n_name
#define n_zeroes        _n._n._n_zeroes
#define n_offset        _n._n._n_offset
#define n_nptr          _n._n_nptr[1]

#define SYMNMLEN        8
#define SYMESZ          18    /* size of a symbol table entry */
```

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 16-34.

Figure 16-35 Auxiliary Symbol Table Entries

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	file name
.text,.data,.bss	C_STAT	DT_NON	T_NULL	section
<i>tagname</i>	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
<i>fname</i>	C_EXT C_STAT	DT_FCN	(Note 1)	function
<i>arrname</i>	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes to Figure 16-34:

1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Figure 16-34, *tagname* means any symbol name including the special symbol *xfake*, and *fname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 16-34 should have a union format in its auxiliary entry.

Note

It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available and should be obtained from the **n_numaux** field in the symbol table.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Figure 16-35.

Figure 16-36 Format for Auxiliary Table Entries for Sections

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	-	-	unused (filled with zeroes)

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 16-36.

Figure 16-37 Tag Names Table Entries

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 16-37:

Figure 16-38 Table Entries for End of Structures

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

The Common Object File Format (COFF)

Functions

The auxiliary table entries for functions have the format shown in Figure 16-38:

Figure 16-39 Table Entries for Functions

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_innoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of function's address in the transfer vector table (not used in the UNIX System)

Arrays

The auxiliary table entries for arrays have the format shown in Figure 16-39. Defining arrays having more than four dimensions produces a warning message.

Figure 16-40 Table Entries for Arrays

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_inno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	-	unused (filled with zeroes)

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 16-40:

Figure 16-41 End of Block and Function Entries

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-17	-	-	unused (filled with zeroes)

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 16-41:

Figure 16-42 Format for Beginning of Block and Function

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 16-42:

Figure 16-43 Entries for Structures, Unions, and Enumerations

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;
struct people
{
    char name[20];
    long id;
};
typedef struct people EMPLOYEE;
```

The symbol **EMPLOYEE** has an auxiliary table entry in the symbol table, but symbol **STUDENT** will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 16-43. This declaration may be found in the header file `syms.h`.

Figure 16-44 Auxiliary Symbol Table Entry (Sheet 1 of 2)

```
union auxent
{
    struct
    {
        long    x_tagndx;
        union
        {
            struct
            {
                unsigned short    x_lnno;
                unsigned short    x_size;
            } x_lnsz;
            long    x_fsize;
        } x_misc;
        union
        {
            struct
            {
                .
                .
                .
            }
        }
    }
}
```

Figure 16-44 Auxiliary Symbol Table Entry (Sheet 2 of 2)

```

    {
        long    x_lnnoptr;
        long    x_endndx;
    } x_fcn;
    struct
    {
        unsigned short    x_dimen[DIMNUM];
    } x_ary;
    } x_fcenary;
    unsigned short    x_tvndx;
} x_sym;
struct
{
    char    x_fname[FILNMLEN];
} x_file;
struct
{
    long    x_scnlen;
    unsigned short    x_nreloc;
    unsigned short    x_nlinno;
} x_scn;
struct
{
    long    x_tvfill;
    unsigned short    x_tvlen;
    unsigned short    x_tvran[2];
} x_tv;
}
#define FILNMLEN    14
#define DIMNUM      4
#define AUXENT      union auxent
#define AUXESZ      18
    
```

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, **long_name_1** and **another_one**) the string table has the format as shown in Figure 16-44:

Figure 16-45 String Table

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

The index of **long_name_1** in the string table is 4 and the index of **another_one** is 16.

Access Routines

UNIX System V/386 Release 3.2 contains a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the *Programmer's Reference*. A summary of what is available can be found in the *Programmer's Reference* under **ldfcn(F)**.

Chapter 17

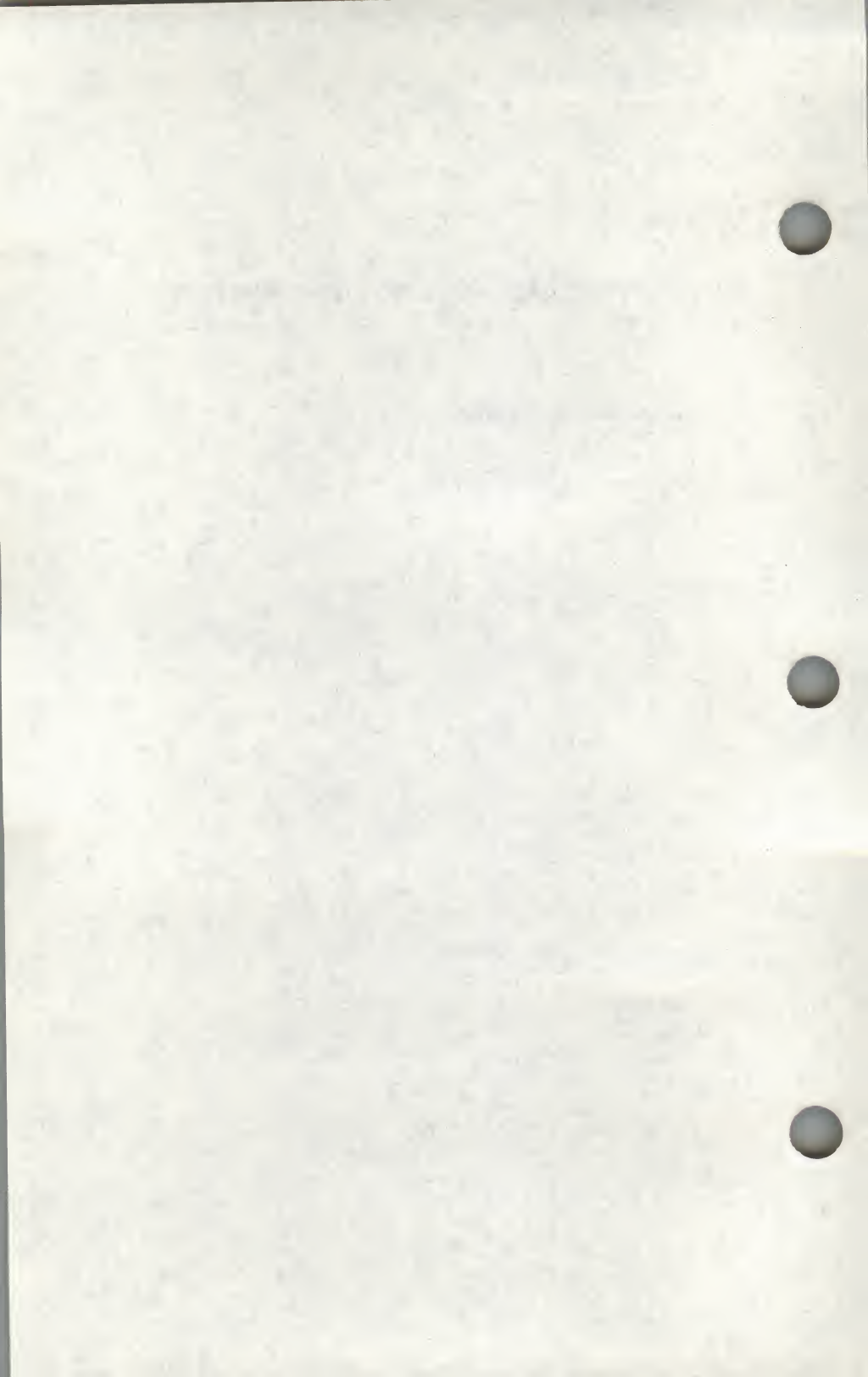
Programming on a Secure System

Programming On A Secure System 17-1

System Calls 17-1

Header Files 17-2

Programming Security 17-2



Programming On A Secure System

This section describes additional system calls and header files available to the programmer working on a secure system.

System Calls

A secure system offers a programmer additional system calls, which control the LUID and system privileges. Note that these calls can make programs non-portable to unsecure and even other secure systems.

The system calls are as follows:

getluid	returns the LUID associated with the current process. This allows a program to check the authenticity of a process' real user ID.
setluid	sets the LUID for a process. In general, the call will not be used by the ordinary user. This is because the LUID cannot be changed once it has been set, and the following commands set the LUID of their processes: login(M) cron(C) and su(C) .
getpriv	returns the calling process's system privileges. The manual page for getpriv(S) describes the arguments and return values that are offered with the call. The data types are defined in the <code><sys/types.h></code> and <code><sys/security.h></code> header files.
setpriv	sets the system privileges for the calling process. The arguments are similar to those for getpriv above. Note that on a secure system you cannot add to the privileges you already possess, you can only remove them.

Header Files

A secure system provides additional header files. These contain definitions and data structures for the security features and are not normally accessed by user programs. The header files are as follows:

- `<sys/security.h>` contains definitions used by security system calls.
- `<prot.h>` includes definitions used by the authentication database routines.
- `<sys/audit.h>` contains audit record definitions and other data structures and definitions associated with the audit subsystem.

Programming Security

- Consult your security administrator before running programs which write sensitive system files, need super-user privilege or manipulate kernel data structures.
- Set up a program's environment and signal handling dispositions. This is essential for programs that execute other programs using the `exec(C)`, `system(C)` and `popen(C)` commands. SUID programs should always set up the IFS and PATH environment variables.
- Restrict umask for programs that create private files.
- Use full pathnames when executing program files.
- Protect programs from general readership.
- Setuid programs which run other programs can be penetrated easily. When passing control to a sub-program, use `setuid(S)` and `setgid(S)` to restore the original permissions. Make sure that all files, which are needed only by the parent process, are closed before invoking the child program.
- Recall that the `access(S)` system call and the `eaccess(S)` system call are different. Use the appropriate one.
- Catch and handle all expected signals. It is important to undo any partially-done actions which may leave sensitive files in an inconsistent state.

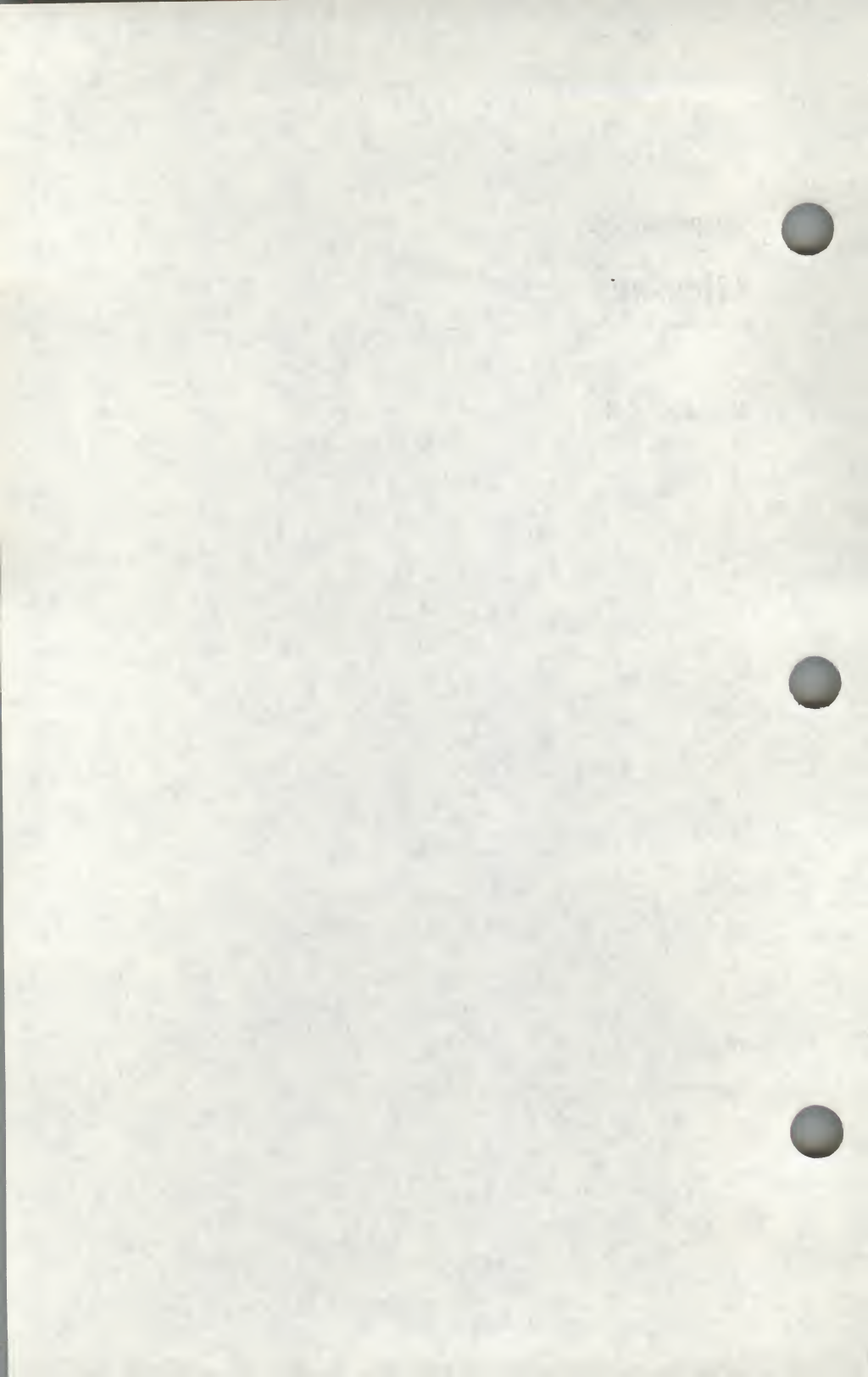
- In all error cases, undo all actions. Remember to check the availability of system resources. For example, not enough processes or out of file system space.
- Handle concurrency. If two users are likely to run a program concurrently, make sure that any files to be modified are locked during the modification.
- Handle errors in file and user input formats. It is possible for any file to be corrupted, and errors which cause your program to fail or leave files or permissions in an inconsistent state invite penetrators.



Appendix A

Glossary

Glossary A-1



Glossary

ANSI standard

ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.

a.out file

a.out is the default file name used by the link editor when it outputs a successfully compiled, executable file. **a.out** contains object files that are combined to create a complete working program. Object file format is described in Chapter 11, "The Common Object File Format," and in **a.out(F)** in the *User's Reference*.

application program

An application program is a working program in a system. Such programs are usually unique to one type of user's work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.

archive

An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command **ar(CP)** collects data for use as a library.

argument

An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments: options and operands. Options are immediately preceded by a minus sign (-) and change the execution or output of the command. Some options can themselves take arguments. Operands are pre-

ceded by a space and specify files or directories that will be operated on by the command. For example, in the command

pr -t -h Heading file

all elements after the **pr** are arguments. **-t** and **-h** are options, **Heading** is an argument to the **-h** option, and **file** is an operand.

For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if more than two are present, they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a page in Section (S) of the *Programmer's Reference*, describes the number and data type of argument(s) expected by the function.

ASCII

ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX System. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.

assembler

The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1 to 1 with the binary machine instructions.

assembly language

A programming language that uses the instruction set that applies to a particular computer.

BASIC

BASIC is a high-level conversational programming language that allows a computer to be used much like a complex electronic calculating machine. The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.

branch table

A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.

buffer

A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents is read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.

byte

A byte is a unit of storage in the computer. On many UNIX Systems, a byte is eight bits (binary digits), the equivalent of one character of text.

byte order

Byte order refers to the order in which data are stored in computer memory.

C

The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. Although it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-processing, and database programs. The C programming language was designed for and implemented on the UNIX System; however, the language is not limited to any one operating system or machine.

- C compiler** The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.
- C preprocessor** The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (#) are directives to the preprocessor. Command line options of the `cc`(CP) command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.
- CCS** CCS is an abbreviation for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, and passes the processed code to the C compiler. The C compiler produces assembly language code that it passes to the assembler. The assembler, in turn, produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an optimizer that makes the code produced as efficient as possible, productivity tools that are used to read and manipulate object files, and libraries that provide runtime support, access to system calls, input/output, string manipulation, mathematical functions, and other code-processing functions.
- COBOL** COBOL is an acronym for COMmon Business Oriented Language. COBOL is a high-level programming language designed for business and commercial applications. The English-language statements of COBOL provide a relatively machine-independent method of expressing a business-oriented problem to the computer.

COFF

COFF is an acronym for Common Object File Format. COFF refers to the format of the output file produced on some UNIX Systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features:

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications.
- Users may make some modifications in the object file construction at compile time.

command

A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.

command line

A command line is composed of the command name followed by any argument(s) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis in a notation designed to show the correct way to type in a command, with or without options and arguments.

compiler

A compiler transforms the high-level language instructions in a program (the source code) into object code or assembly language. Assembly language code may then be passed to the assembler for further translation into machine instructions.

core

Core is a (mostly archaic) synonym for primary memory.

Glossary

A

core file

A core file is an image of a terminated process saved for debugging. A core file is created under the name "core" in the current directory of the process when an abnormal event occurs resulting in the process' termination. A list of these events is found in the **signal(S)** manual page in Section (S) of the *Programmer's Reference*.

core image

Core image is a copy of all the segments of a running or terminated program. The copy may exist in main storage, in the swap area, or in a core file.

curses

curses(S) is a library of C routines that are designed to handle input, output, and other operations in screen management programs. The name **curses** comes from the cursor optimization that the routines provide. When a screen management program is run, cursor optimization minimizes the amount of time a cursor has to move about a screen to update its contents. The program refers to the **terminfo(F)** database at run time to obtain the information that it needs about the screen (terminal) being used. See **terminfo(F)** in the *User's Reference*.

data symbol

A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.

database

A database is a bank of information on a particular subject or subjects. On-line databases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.

debug

Debugging is the process of locating and correcting errors in computer programs.

default

A default is the way a computer will perform a task in the absence of other instructions.

delimiter

A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX System command syntax, a minus sign (-) is most often the delimiter for option names, for example, **-s** or **-n**, although some commands also use a plus sign (+).

directory

A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization.

The **ls(C)** command is used to list the contents of a directory. When you first log onto the system, you are in your home directory (**\$HOME**). You can move to another directory by using the **cd(C)** command and you can print the name of the current directory by using the **pwd(C)** command. You can also create new directories with the **mkdir(C)** command and remove empty directories with **rmdir(C)**.

A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative path name or the full path name of a directory.

dynamic linking

Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.

environment

An environment is a collection of resources used to support a function. In the UNIX System, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

An environment variable is a shell variable such as \$HOME (which stands for your login directory) or \$PATH (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to work. These variables come from `/etc/profile`, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal `.profile` file, which you create in your login directory to tailor your own working environment. You can also temporarily set variables at the shell level.

executable file

An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the `chmod(C)` command. In addition to being ready to run, a file in the UNIX System needs to have execute permission.

exit

Exit is a specific system call that causes the termination of a process. The `exit(S)` call will close any open files and clean up most other information and memory which was used by the process.

- exit status: return code**
An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.
- exported**
An exported symbol is a symbol that a shared library defines and makes available outside the library. See imported symbol.
- expression**
An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.
- file**
A file is an identifiable collection of information that, in the UNIX System, is a member of a file system. A file is known to the UNIX System as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs, or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special, or directory files.
- file and record locking**
The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.
- file descriptor**
A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).

Glossary

A

file system

A UNIX System file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.

filter

A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are **grep(C)** and **tail(C)**, which select and output part of the input; **sort(C)**, which sorts the input; and **wc(C)**, which counts the number of words, characters, and lines in the input. **sed(C)** and **awk(C)** are also filters but they are called programmable filters or data transformers because, in addition to the data to be transformed, a program must be supplied as input.

flag

A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX System flags are usually indicated by a leading hyphen (-). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean "to point out" or "to draw attention to". See option.

fork

fork(S) is a system call that divides a new process into two processes, the parent process and the child processes, with separate, but initially identical, text, data, and stack segments. After duplication, the child (created) process is given a return code of 0 and the parent process is given the process id of the newly created child as the return code.

FORTTRAN	FORTTRAN is an acronym for FORM ula TRAN slator. It is a high-level programming language originally designed for scientific and engineering calculations but is now widely adapted for many business uses also.
function	A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.
header file	A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.
high-level language	A high-level language is a computer programming language such as C, FORTTRAN , COBOL , or PASCAL that uses symbols and command statements representing actions the computer is to perform, the exact steps for a machine to follow. A high-level language must be translated into machine language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of machine language instructions. Low-level details of the computer's internal organization are left to the compilation system.
host machine	A host machine is the machine on which an a.out file is built.
imported symbol	An imported symbol is a symbol used but not defined by a shared library. See exported symbol.

interpreted language

An interpreted language is a high-level language that is not either translated by a compilation system or stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.

Interprocess Communication

Interprocess Communication describes software that enables independent processes running at the same time to exchange information through messages, semaphores, or shared memory.

interrupt

An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX System programs by pressing the delete or break keys, by typing **CTRL-D**, or by using the **kill(C)** command.

I/O (Input/Output)

I/O is the process by which information enters (input) and leaves (output) the computer system.

kernel

The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX System relies. It is responsible for most operating system functions. It schedules and manages work done by the computer and maintains the file system. The kernel has its own text, data, and stack areas.

lexical analysis

Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as `=`, `:=`, and `;`. Lexical analysis enables you to recognize, for example, that the stream of characters `'print("hello, universe")'` is to be analyzed into a series of tokens beginning with

the word 'print' [not with the string 'print("h.'). In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).

- library** A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.
- link editor** A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX System. You can also link edit previously compiled files by using the **ld**(CP) command.
- magic number** The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether it is made up of shared or non-shared text, and on which processor the file is executable.
- makefile** A makefile is a file that lists dependencies among the source-code files of a software product and methods for updating them, usually by recompilation. The **make**(CP) command uses the makefile to maintain self-consistent software.
- manual page** A manual page, or "man page" in UNIX System jargon, is the repository for the detailed description of a command, a system call, a subroutine, or some other UNIX System component.
- null pointer** A null pointer is a C pointer with a value of 0.

Glossary

object code

Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file that is ready to run is an executable file.

optimizer

An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by, and speeds the execution time of, the code.

option

An option is an argument used in a command line to modify program output by modifying the execution of a command. It is usually one character preceded by a hyphen (-). When you do not specify any options, the command will execute according to its default options. For example, in the command line

ls -a -l directory

-a and **-l** are the options that modify the **ls(C)** command to list all **directory** entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).

parent process

A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.

parse

To parse is to analyze a sentence in order to identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are analyzed.

PASCAL

PASCAL is a multipurpose high-level programming language often used to teach programming. It is based on the ALGOL programming language and emphasizes structured programming.

path name

A path name is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the path name begins with a slash, it is called a full path name; the initial slash means that the path begins at the root directory.

A path name that does not begin with a slash is known as a relative path name, meaning relative to the present working directory. A relative path name may begin either with a directory name or with two dots followed by a slash (../). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with ../ indicates that the path first proceeds up the hierarchy; ../ is the parent of the present working directory.

permissions

Permissions are a means of defining a right to access a file or directory in the UNIX System file system. They are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions:

- ☐ Read permission (r) includes permission to cat, pg, lp, and cp a file.
- ☐ Write permission (w) is the permission to change a file.
- ☐ Execute permission (x) is the permission to run an executable file.

Permissions can be changed with the **chmod(C)** command (see the *User's Reference*).

pipe

A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe symbol (|), which indicates that the output from the process on the left should be routed to the process on the right. For

example, in the command

who | wc -l

the output from the **who(C)** command, which lists the users who are logged on to the system, is used as input for the word-count command, **wc(C)**, with the **l** option. The result of this pipe-line (succession of commands connected by pipes) is the number of people who are currently logged on to the system.

portability

Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other systems, time is wasted, and each modification increases the chances for an error.

preprocessor

Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn** and **tbl** are preprocessors for **nroff**. **grap** is a preprocessor for **pic**. **cpp(CP)** is a preprocessor for the C compiler.

process

A process is a program that is at some stage of execution. In the UNIX System, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines.

The process id is a unique system-wide identification number that identifies an active process. The process status command, **ps(C)**, prints the process ids of the processes that belong to you.

program	A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting system status. A sub-program is part of a larger program and can be compiled independently.
regular expression	A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to be searched for in a file. The pattern-matching functions of <code>ed(C)</code> and <code>grep(C)</code> , for example, use regular expressions.
routine	A routine is a discrete section of a program to accomplish a set of related tasks
semaphore	In the UNIX System, a semaphore is a sharable short unsigned integer maintained through a family of system calls which include calls for increasing the value of the semaphore, setting its value, and for blocking waiting for its value to reach some value. Semaphores are part of the UNIX System IPC facility.
shared library	Shared libraries include object modules that may be shared among several processes at execution time.
shared memory	Shared memory is an IPC (interprocess communication) facility in which two or more processes can share the same data space.
shell	The shell is the UNIX System program— <code>sh(C)</code> —responsible for handling all interaction between you and the system. It is a command language interpreter that understands your commands and causes the computer to act on them. The shell also establishes the environment at your terminal. A shell normally is started for you as part of the login process. Three shells, the Bourne shell, the Korn shell, and the C shell, are popular. The shell can also be used as a programming language to write procedures for a variety of tasks.

Glossary

signal: signal number

A signal is a message that you send to processes or processes send to one another. The most common signals you might send to a process are ones that would cause the process to stop: for example, interrupt, quit, or kill. A signal sent by a running process is usually a sign of an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.

source code

Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.

standard error

Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX System, the default case is to associate standard error with the user's terminal.

standard input

Standard input is an input stream to a program. In the UNIX System, the default case is to associate standard input with the user's terminal.

standard output

Standard output is an output stream from a program. In the UNIX System, the default case is to associate standard output with the user's terminal.

stdio: standard input-output

stdio(S) is a collection of functions for formatted and character-by-character input/output at a higher level than the basic read, write, and open operations.

static linking

Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.

- stream**
- A stream is an open file with buffering provided by the `stdio` package.
 - A stream is a full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character input/output interface for user processes.
- string**
- A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.
- strip**
- strip(CP)** is a command that removes the symbol table and relocation bits from an executable file.
- subroutine**
- A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.
- symbol table**
- A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.
- symbol value**
- The value of a symbol, typically its virtual address, is used to resolve references.

syntax

- Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command.
- Language syntax is the set of rules that describes how the elements of a programming language may legally be used.

system call

A system call is a request by an active process for a service performed by the UNIX System kernel, such as I/O, process creation, etc. All system operations are allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid(S)** system call allows you to inspect information that is not normally available, since it resides in the operating system's address space.

target machine

A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.

TCP/IP (Transmission Control Protocol/Internetwork Protocol)

TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.

terminal definition

A terminal definition is an entry in the **terminfo(F)** database that describes the characteristics of a terminal. See **terminfo(F)** and **curses(S)** in the *User's* and *Programmer's References*, respectively.

terminfo

- **terminfo** is a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys.
- **terminfo** is a database containing the compiled descriptions of many terminals that can be used with **curses(S)** screen management programs. These descriptions specify the capabilities of a terminal and how it performs various operations (— for example), how many lines and columns it has, and how its control characters are interpreted. A **curses(S)** program refers to the database at run time to obtain information it needs about the terminal being used.

See **curses(S)** in the *Programmer's Reference*. **terminfo(F)** routines can be used in shell programs, as well as C programs.

text symbol

A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.

tool

A tool is a program, or package of programs, that performs a given task.

trap

A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX Operating System

The UNIX Operating System is a general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T. An operating system is the software on the computer under which all other software runs. The UNIX Operating System has two basic parts:

- The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.

A

- The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

The utility programs or UNIX System commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.

userid

A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.

utility

A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software errors, for example, a loader, editor, debugging, or diagnostics package.

variable

- A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next.
- A variable in the shell is a name representing a string of characters (a string value).
- A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter).
- A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.

white space

White space is one or more spaces, tabs, or new-line characters. White space is normally used to separate strings of characters and is required to separate the command from its arguments on a command line.

window

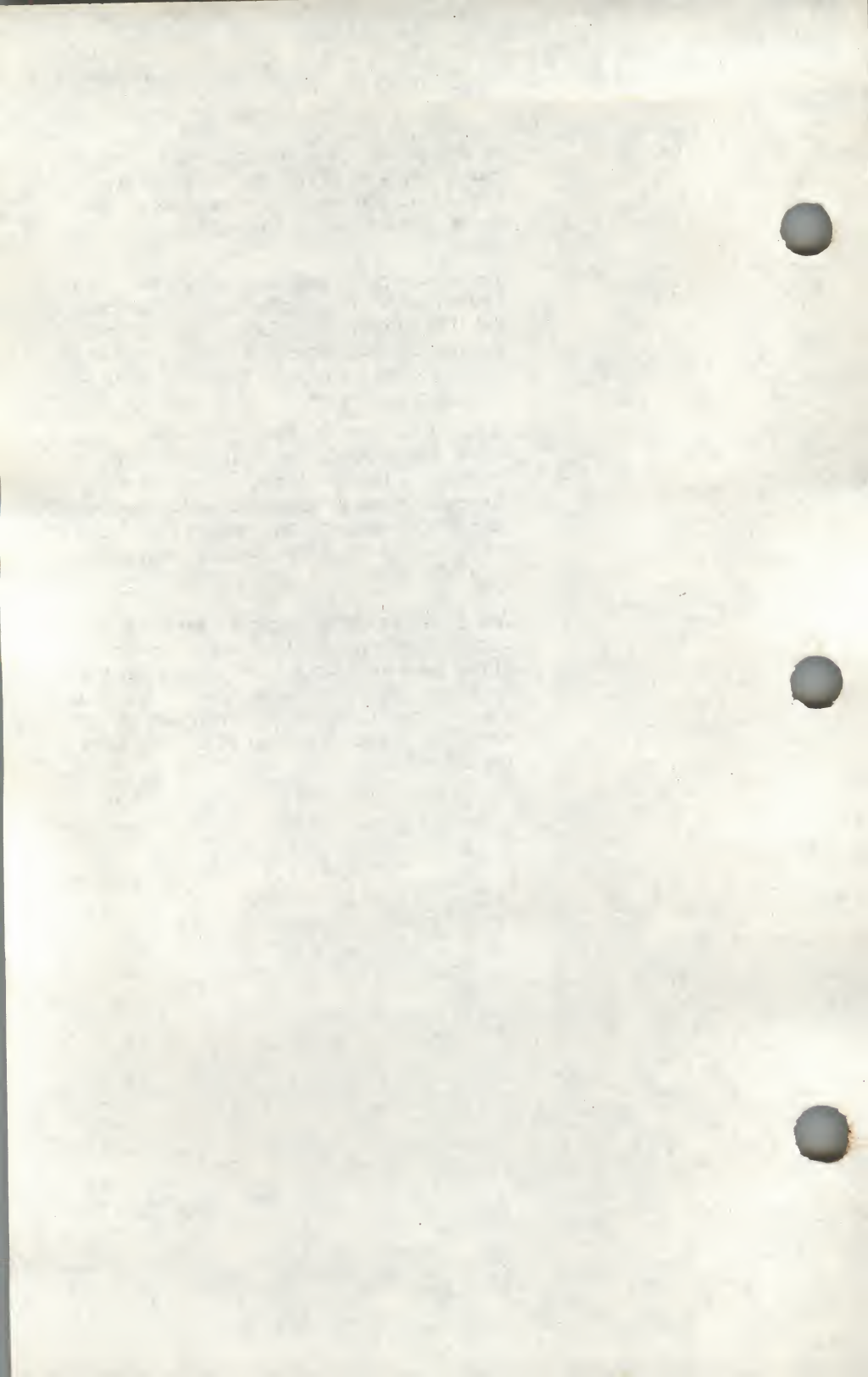
A window is a screen within your terminal screen that is set off from the rest of the screen. If you have two windows on your screen, they are independent of each other and the rest of the screen.

The most common way to create windows on a UNIX System is by using the layers capability of the TELETYPE 5620 Dot-Mapped Display. Each window you create with this program has a separate shell running it. Each one of these shells is called a layer.

If you do not have this facility, the `shl(C)` command, which stands for shell layer, offers a function similar to the layers program. You cannot create windows using `shl(C)`, but you can start different shells that are independent of each other. Each of the shells you create with `shl(C)` is called a layer.

word

A word is a unit of storage in a computer that is composed of bytes of information. The number of bytes in a word depends on the computer you are using. The 80286 Computer has 16 bits or 2 bytes per word. The 80386 Computer has 32 bits or 4 bytes per word, and 16 bits or 2 bytes per half word.



Index

A

Accessing Values in Enclosing Rules 4-36

Actions 4-6

adb

- ? and / commands 12-15

- = command 12-14

- backtrace 12-23

- binary files 12-48

- breakpoints 12-20

- combining commands on a single line 12-41

- computing numbers and

 - displaying text 12-44

- debugging 12-2

- core image files 12-3

- creating scripts 12-41

- current address 12-9

- data files 12-4

- data formats 12-12

- decimal integers 12-7

- deleting breakpoints 12-22

- displaying CPU registers 12-23

- displaying external variables 12-24

- displaying instructions and data 12-6

- exiting 12-5

- forming addresses 12-6

- forming expressions 12-7

- hexadecimal integers 12-7

- introduction 12-1

- killing a program 12-22

- leaving 12-5

- locating values in a file 12-48

- making changes to memory 12-49

- memory maps 12-35

- miscellaneous features 12-41

- octal integers 12-7

- operators 12-11

- patching binaries 12-48

- program execution 12-18

- prompt option 12-4

- register names 12-10

- setting default input format 12-43

- setting output width 12-42

- setting the maximum offset 12-42

- single-stepping a program 12-22

- starting and stopping 12-2

- symbols 12-7

- using UNIX commands 12-44

- validating addresses 12-40

adb (*continued*)

- variables 12-8

- write option 12-4

- writing to a file 12-49

Adding an Argument to a Function 9-21

Addresses 6-2

admin Command 7-27

Advisory Locking 13-16

Aligning an Output Section 6-11

Allocating a Section Into Named Memory 6-17

Allocation Algorithm 6-24

Ambiguity and Conflicts 4-17

a.out 14-2, 14-3, 14-8, 14-9, 14-10, 14-12, 14-22, 14-29, 14-33, 14-35, 14-43, 14-50

ar 1-3

Archive Libraries 5-13

Archive library 14-2, 14-3, 14-5, 14-6, 14-8, 14-9, 14-10, 14-25, 14-29

- compatibility 14-36

Arguments, macro 8-7

Assembler, basic tool 1-2

Assignment Statements 6-5

Assignments of longs to ints 3-10

Attaching a Shared Memory Segment 15-82

Auditing 7-39

B

Basic Features 5-2

Basic Specifications 4-4

Binding 6-2

Branch table 14-10, 14-23, 14-26

Branch table specifications 14-17

Breakpoints, setting and deleting 11-9

C

C programming language 1-1

Calling Functions 11-11

Caveat Emptor—Mandatory Locking 13-17

cdc Command 7-33

cflow 14-38

changequote function 8-6

chkshlib 14-40, 14-41, 14-42, 14-43

comb Command 7-35

Command

- execution 1-5

- interpretation 1-5

Index

- Command Line Syntax for Editors 9-23
- Command Usage 5-20
- Comments 5-6
- Concurrent Edits of Different SID 7-18
- Concurrent Edits of Same SID 7-21
- Conditionals
 - m4 macro processor 8-12
- Continuation Lines 5-6
- Cross-Reference File 9-5
- cscope 9-3, 9-6, 9-16, 9-17, 9-22
- csd 1-5
- C-shell
 - command history mechanism 1-5
 - command language 1-5

D

- Data File Cannot Be Found 9-39
- Data symbol 14-25
- Deadlock Handling 13-15
- Dealing With Holes in Physical Memory 6-23
- define function 8-3
- Defining macros 8-3
- delta Command 7-24
- Delta Numbering 7-7
- dempdef function 8-15
- Dependency Information 5-7
- Description Files and Substitutions 5-6
- Detaching Shared Memory Segments 15-83
- Displaying Machine Language Statements 11-11
- divert function 8-10
- divnum function 8-10
- dnl function 8-14
- DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections 6-26
- dump 14-12, 14-41
- Dumping Core 9-22
- Dynamic Dependency Parameters 5-19

E

- Environment Variables 5-22
- Error Handling 4-27
- Error message file
 - creation 1-4
- Error Messages 7-11
- eval function 8-8
- Examining Variables 11-3

- Example Program 15-10, 15-15, 15-23, 15-41, 15-46, 15-57, 15-70, 15-75, 15-83
- Executable Commands 5-12
- Expressions 6-4
- Extensions of \$)**\$, \$@, and \$< 5-13
- External symbol 14-25

F

- Failure of Data to Merge 9-37
- File
 - archives 1-3
 - error message file *See* Error message file
 - protection 13-4
 - setting a lock 13-5
 - specifications 6-9
- Flow of Control 3-6
- Formatting 7-38
- Function Values 3-6

G

- get Command 7-13
- Global data 14-24, 14-26, 14-48
- Grouping Sections Together 6-12

H

- help Command 7-6
- Hints for Preparing Specifications 4-32
- Host shared library 14-9, 14-19, 14-20, 14-33, 14-35, 14-42, 14-43

I

- ID Keywords 7-14
- ifdef function 8-6
- ifndef function 8-12
- Implicit Rules 5-10
- Imported symbol 14-34
- Improving Performance with prof and lprof 9-40
- Improving Test Coverage with lprof 9-47
- include Files 5-18

- include function 8-10
 - sininclude function 8-10
- incr function 8-8
- Incremental Link Editing 6-24
- Initialized Section Holes or .bss Sections 6-18
- Input Style 4-32
- Inserting Commentary for the Initial Delta 7-28
- Internal Rules 5-25
- Interpreting Profiling Output 9-31
- Introducing the C Programmer's Productivity Tools 9-1

K

- Keeping Data Files in a Separate Directory 9-29
- Key Letters That Affect Output 7-22

L

- ld
 - basic tool 1-2
- Left Recursion 4-33
- len function 8-13
- lex
 - description 1-2
- Lexical Analysis 4-9
- Lexical Tie-Ins 4-34
- .lib 14-2
- Library
 - archive 14-2
 - conversion 1-3
 - host 14-19, 14-33
 - maintenance 1-3
 - members 14-22
 - networking 14-3, 14-4, 14-15, 14-16
 - shared 14-2, 14-3
 - specification file 14-16, 14-17
 - target 14-16, 14-19
- Link Editor 6-1
- Link Editor Command Language 6-4
- lint Message Types 3-4
- Load a Section at a Specified Address 6-10
- Loader *See* ld
- Lock Information
- lprof 9-25, 9-36

M

- m4
 - arithmetic values 8-8
 - basic operation 8-1
 - changequote function 8-6
 - define function 8-3
 - dempdef function 8-15
 - description 1-2, 8-1
 - divert function 8-10
 - divnum function 8-10
 - dnl function 8-14
 - eval function 8-8
 - ifdef function 8-6
 - ifelse function 8-12
 - include function 8-10
 - incr function 8-8
 - invoking macro processor 8-2
 - len function 8-13
 - maketemp function 8-11
 - overview 8-1
 - printing function 8-15
 - sininclude function 8-10
 - substr function 8-13
 - syscmd function 8-11
 - translit function 8-13
 - undefine function 8-6
 - undivert function 8-10
 - using arguments 8-7
- m4 *See Also* Macro processor
- Machine Language Debugging 11-11
- Macro Definitions 5-6
- Macro processor
 - arguments 8-7
 - arithmetic values 8-8
 - built-in function
 - changequote 8-6
 - define 8-3
 - dempdef 8-15
 - divert 8-10
 - divnum 8-10
 - dnl 8-14
 - eval 8-8
 - ifdef 8-6
 - ifelse 8-12
 - include 8-10
 - incr 8-8
 - len 8-13
 - maketemp 8-11
 - printing 8-15
 - sininclude 8-10
 - substr 8-13
 - syscmd 8-11

Index

Macro processor (*continued*)
 built-in function (*continued*)
 translit 8-13
 undefine 8-6
 undivert 8-10
 conditionals 8-12
 defining macros 8-3
 functions 8-1
 m4 8-1
 manipulating strings 8-13
 printing 8-15
 quoting 8-5
 redirecting input 8-2
 redirecting output 8-2
 removing functions 8-6
 standard input 8-2
 standard output 8-2
 system commands 8-11
Macros 8-1
 built-in 8-1
 preprocessing 1-2
 user defined 8-1
main 14-33
Maintainer *See* Make
Make
 basic tool 1-4
make Command 5-20
maketemp function 8-11
malloc 14-30, 14-31, 14-33
Manipulating files
 include function 8-10
 macro processors 8-10
 using m4 to 8-10
Mandatory Locking 13-16
Manipulating Registers 11-12
Manipulating strings
 m4 macro processor 8-13
Memory Configuration 6-1, 6-7
Merging Data Files 9-29
Merging Option 9-35
Messages 15-2, 15-21
Message Queues 15-7, 15-13
mkshlib 14-1, 14-13, 14-16, 14-19, 14-20, 14-35,
 14-49
msgctl 15-14
msgget 15-7
msgop 15-21
msgrcv 15-25
msgsnd 15-24

N

Nonportable Character Use 3-9
Nonrelocatable Input Files 6-28
Non-Terminating Programs 9-36
Notational conventions 1-8
Notes and Special Considerations 6-20
Null Suffix 5-18

O

Object file 6-3, 14-18
Old Syntax 3-11
Opening a File for Record Locking 13-4
Operations for Messages 15-20
Operations for Shared Memory 15-81
Operations on Semaphores 15-55
Optional Features 9-16
Output File Blocking 6-27
Output Translations 5-7

P

Paging 14-37, 14-38
Parser Operation 4-12
Pointer Alignment 3-12
Precedence 4-23
printf 14-4, 14-23
Printing
 m4 macro processor 8-15
Printing a Stack Trace 11-3
printing function 8-15
Profiling Examples 9-40
Profiling Programs that Fork 9-30
Profiling within a Shell Script 9-30
PROFOPTS Environment Variable 9-27, 9-28
Program
 maintainer *See* Make
 testing 11-8
Program development 1-1
Programming On A Secure System 17-1
Programming Security 17-2
Protection 7-37
prs Command 7-30

R

ranlib
 description 1-3
 Receiving Messages 15-22
 Record Locking and Future Releases of UNIX
 13-17
 Record Locks 13-7
 Recording Changes by Means of delta 7-4
 Recursive Makefiles 5-9
 Reserved Words 4-35
 Retrieval of Different Versions 7-14
 Retrieval With Intent to Make a Delta 7-16
 Retrieving a File by Means of get 7-3
 rmdel Command 7-32

S

sact Command 7-32
 Saving space 14-6
 SCCS 1-4
 Command Conventions 7-10
 Commands 7-12
 Creation 7-27
 Files 7-37
 for Beginners 7-2
 Initialization and Modification of parameters
 Makefiles 5-19
 using admin
 scscdiff Command 7-34
 sdb 11-2, 14-12
 Section Definition Directives 6-8
 Sections 6-2
 Semaphores 15-32, 15-34, 15-37, 15-45
 semctl 15-45
 semget 15-37
 semop 15-55
 Set/Used Information 3-5
 Shared library 14-1, 14-2, 14-3, 14-4, 14-5,
 14-6, 14-7, 14-8, 14-9, 14-10, 14-11, 14-12,
 14-13, 14-15, 14-16, 14-17, 14-18, 14-19,
 14-20, 14-22, 14-23, 14-24, 14-25, 14-26,
 14-27, 14-28, 14-29, 14-30, 14-31, 14-32,
 14-34, 14-35, 14-37, 14-39, 14-40, 14-48,
 14-49, 14-50, 14-51, 14-52
 branch table 14-11
 building 14-13, 14-51
 compatibility 14-40
 contents 14-16
 debugging 14-12
 example 14-44
 Shared library (*continued*)
 global data 14-24
 guidelines 14-21
 source files 14-45
 space 14-6, 14-11
 Shared Memory 15-62, 15-63, 15-67, 15-73
 shkshlib 14-40
 shlib 14-16
 shmat 15-84
 shmctl 15-74
 shmdt 15-85
 shmget 15-67
 shmop 15-82
 Simulating error and accept in Actions 4-36
 sinclude function 8-10
 Software development
 described 1-1
 Source code 14-5
 Source file 14-45, 14-48
 Source File
 Display and Manipulation 11-6
 Source file
 shared library 14-45, 14-46, 14-47
 Source Files in a Different Directory 9-34
 Source Listing for a Subset of Files 9-34
 Source Listing Option 9-31
 Specification file 14-26, 14-50
 Specifying Program Names 9-37
 Specifying a Program and Data File 9-31
 lprof on lprof 9-41
 Shared Libraries 9-39
 Stacking cscope and Editor Calls 9-16
 Static data 14-23
 Strange Constructions 3-10
 strcmp 14-11
 substr function 8-13
 Suffixes and Transformation Rules 5-9
 Summary Option 9-34
 Support for Arbitrary Value Types 4-37
 Symbol 14-29
 data 14-25
 external 14-25, 14-27
 imported 14-28, 14-34
 static library 14-25
 Syntax Diagram for Input Directives 6-30
 syscmd function 8-11
 System Calls 17-1
 System commands
 m4 macro processor 8-11

Index

T

- Table
 - branch 14-10
- Tags file
 - creation 1-4
- Target library 14-9, 14-16, 14-19, 14-29, 14-38, 14-42, 14-43
- Target shared library 14-20
- Terminology 7-2, 13-2
- Tilde in SCCS Filenames 5-16
- translit function 8-13
- Trouble at Compile Time 9-36
- Trouble at the End of Execution 9-38
- Turning Off Profiling 9-28
- Type Casts 3-9
- Type Checking 3-8

U

- undefine function 8-6
- undivert function 8-10
- Undoing a get -e 7-18
- UNIX Timesharing system 1-1
- Unknown Terminal Type 9-22
- Unused Variables and Functions 3-4
- Usage 3-2
- Use of Archive Libraries 6-20

V

- val Command 7-36
- vc Command 7-36
- Viewpaths 9-16

W

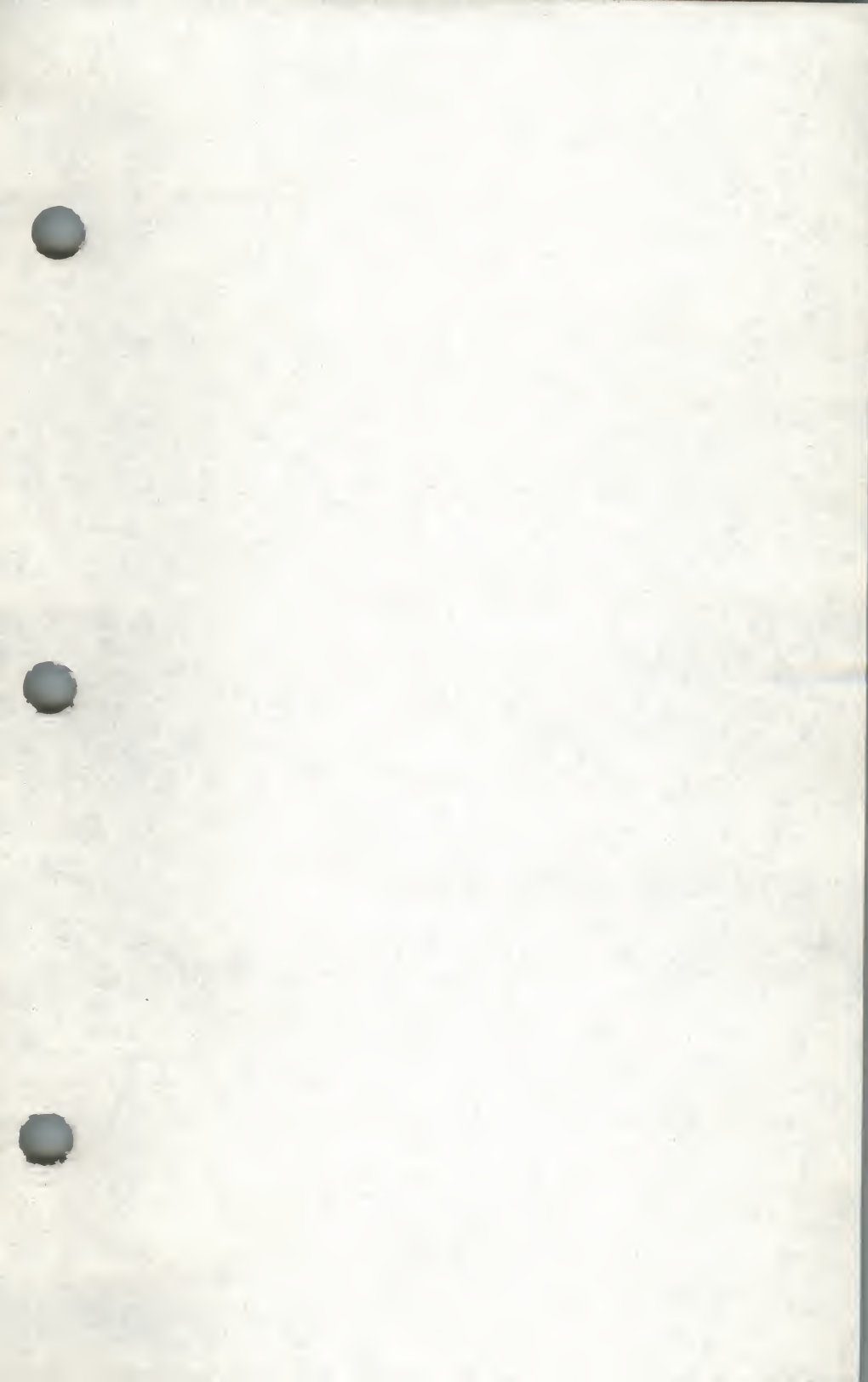
- what Command

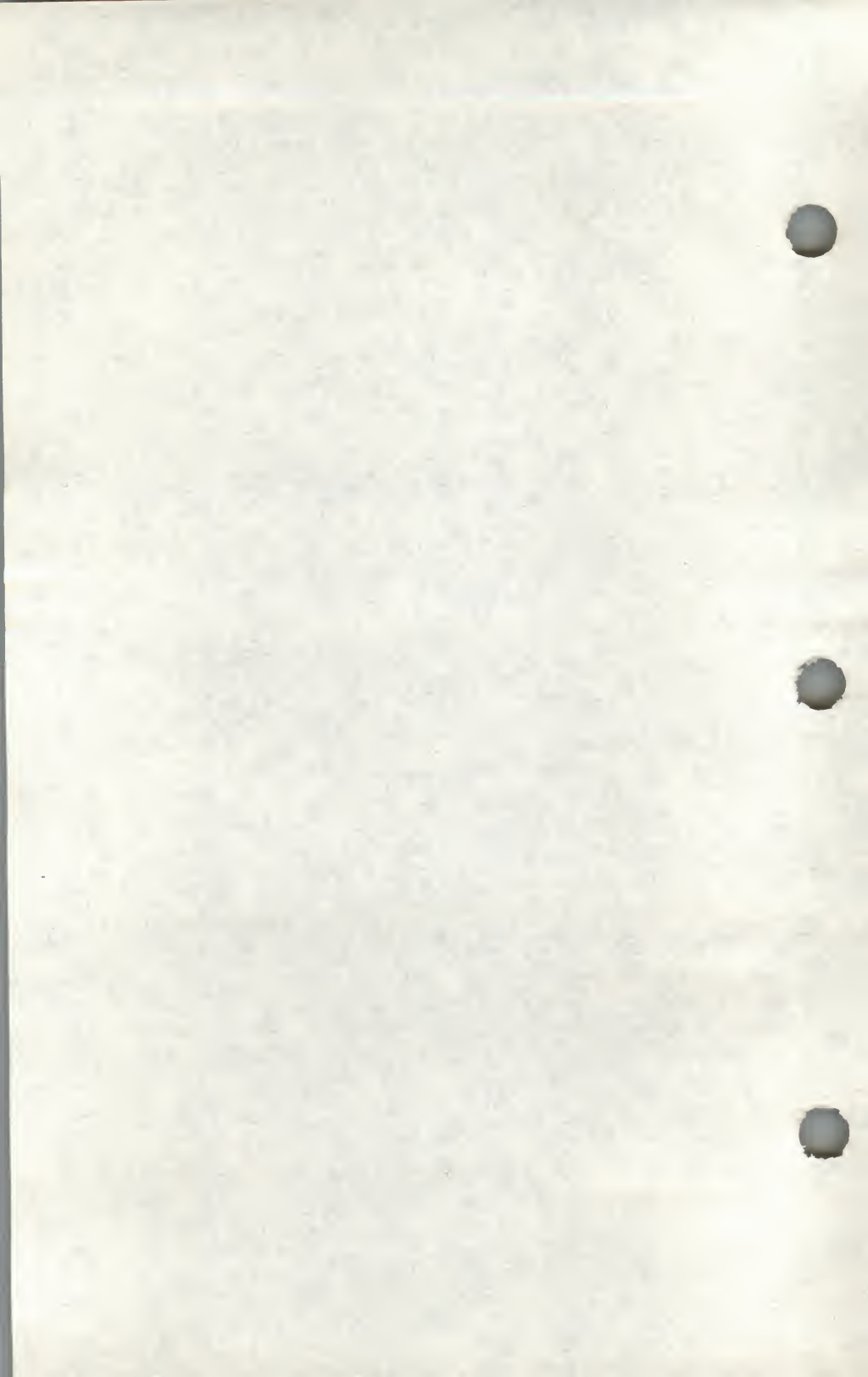
X

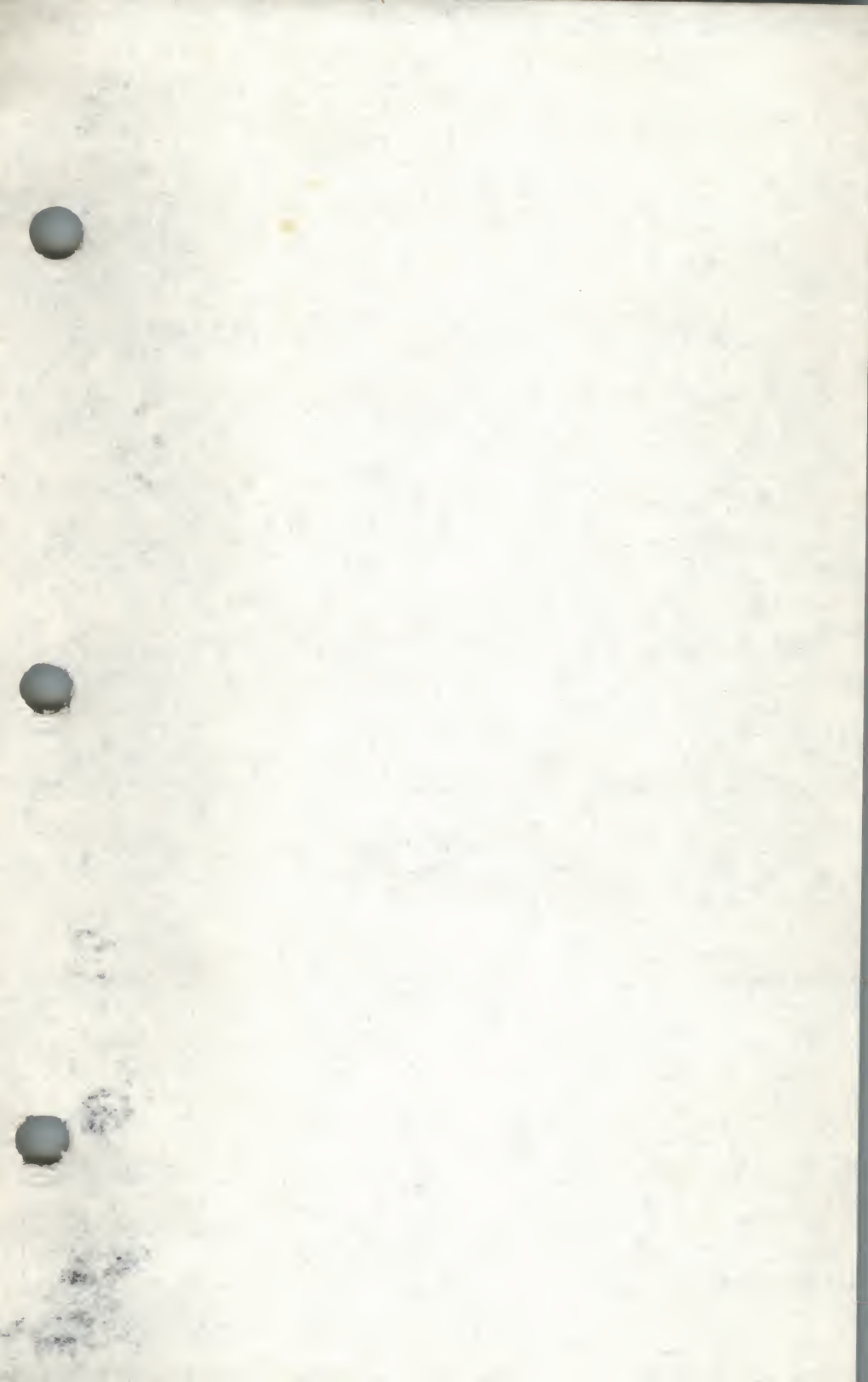
- x.files and z.files 7-11

Y

- yacc
 - description 1-2
 - yacc Environment 4-30
 - yacc Input Syntax 4-39







514-210-900
23335